More Information

Textbook

Others

Solving Recurrences

Examples:
- \( T(n) = 2T(n/2) + \Theta(n) \)
- \( T(n) = \Theta(n \log n) \)
- \( T(n) = 2T(n/2) + n \)
- \( T(n) = \Theta(n \log n) \)

Three methods for solving recurrences
- Substitution method
- Iteration method
- Master method

Recurrence Examples

\[
T(n) = \begin{cases} 
0 & n = 0 \\
0 & n > 0 \\
c + T(n-1) & n > 0 
\end{cases}
\]

\[
T(n) = \begin{cases} 
0 & n = 0 \\
0 & n > 0 \\
c + T(n-1) & n > 0 
\end{cases}
\]

Substitution Method

The substitution method
- “Making a good guess method”
- Given the form of the answer, then
- Use induction to find the constants and show that solution works

Our goal: show that \( T(n) = 2T(n/2) + n = O(n \log n) \)
Thus, we need to show that $T(n) \leq cn \log n$ with an appropriate choice of $c$.

- Inductive hypothesis: assume $T(n/2) \leq cn \log (n/2)$.
- Substitute back into recurrence to show that $T(n) \leq cn \log n$ holds, when $r \geq 1$.

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

Thus in general $T(n) = O(n \log n)$.

Consider $T(n) = 2T(\sqrt{n}) + \log n$.

- Simplify it by letting $x = \sqrt{n}$.
- Rename $S(x) = T(2^x)$.
- Changing back from $S(x)$ to $T(n)$, we obtain $T(n) = T(2^n)$.

$$T(2^n) = 2T(2^{n/2}) + m$$

$$S(x) = O(x \log x)$$

$$= O(n \log \log n)$$

Iteration Method

- Iteration method:
  - Expand the recurrence $k$ times.
  - Work some algebra to express as a summation.
  - Evaluate the summation.

$$T(n) = \begin{cases} 0 & n = 0 \\ c + T(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} 0 & n = 0 \\ c + T(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} 0 & n = 0 \\ c + T(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} 0 & n = 0 \\ c + T(n-1) & n > 0 \end{cases}$$

$$T(n) = \begin{cases} 0 & n = 0 \\ c + T(n-1) & n > 0 \end{cases}$$
Design and Analysis of Algorithms

Chapter 3

\[ T(n) = \begin{cases} \frac{C}{2T\left(\frac{n}{2}\right)} + c & n \leq 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases} \]

1. So far for \( n \geq k \) we have
   - \( T(n) = 2T(n/2) + (2^k - 1)c \)
2. To stop the recursion, we should have
   - \( n^k = 1 \Rightarrow k = \log n \)
   - \( T(n) = 2^k T(n/2^k) + (2^k - 1)c \)
   - \( n T(n) + (n - 1)c \)
   - \( n T(1) + (n - 1)c \)
   - \( n c + (n - 1)c = 2n - c = cn - c/2 \)
   - \( \leq cn = O(n) \) for all \( n \geq \frac{1}{8} \)

So with \( k = \log_n n \)
- \( T(n) = cn (a^k b^0 + \ldots + a^k b^0 + a b + 1) \)

What if \( a = b \)?
- \( T(n) = cn(1 + \ldots + 1 + 1) \) \( \because k+1 \) times
  - \( cn(k + 1) \)
  - \( cn \log_n n + 1 \)
  - \( \Theta(n \log_n n) \)

So we have
- \( T(n) = aT(n/b) + cn(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)
- \( n b^0 = 1 \Rightarrow n = b^0 \Rightarrow k = \log_n n \)
- \( T(n) = a^k T(1) + cn(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)
- \( = a^k c + cn(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)
- \( = cn(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)
- \( \leq cn(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)

\[
\sum_{i=0}^{a-1} a^i b^i + \cdots + a^i b^i + \cdots + a^i b^i = \frac{(a^i b^i)^{k+1} - 1}{a^i b^i - 1} = \frac{1}{a^i b^i - 1} - 1
\]

\[ T(n) = aT\left(\frac{n}{b}\right) + cn \quad n > 1 \]

So with \( k = \log_n n \)
- \( T(n) = cn(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)

What if \( a > b \)?
- \( T(n) = cn \cdot \Theta(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)
- \( \frac{a^1}{a^0} + \frac{a^2}{a^1} + \cdots + a^a}{a^0} = \frac{(a^i b^i)^{k+1} - 1}{(a^i b^i - 1)} = \Theta(a^i b^i) \)
- \( T(n) = cn \cdot \Theta(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)
- \( = cn \cdot \Theta(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)
- \( \leq \Theta(a^k b^0 + \ldots + a^k b^0 + a b + 1) \)
So...

where applies:

case $T$

since $T$

Thus the solution is

$T$

case $a$

then

Given: a

Then, the Master Theorem gives us a cookbook for the algorithm’s running time:

The Master Theorem

if $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ & $b > 1$

then

The Master Method Case 1

$T(n) = nT(n/3) + n$

where $a = 9$, $b = 3$, $f(n) = n$

& since $9^{(\log_3 n) - 1} = n^{\log_3 9 - 1} = n^{0.5}$

Thus the solution is $T(n) = \Theta(n^{0.5})$

Using The Master Method Case 2

$T(n) = 2T(n/3) + 1$

where $a = 2$, $b = 3$, $f(n) = 1$

& since $2^{(\log_3 n) - 1} = \Omega(1)$

Thus the solution is $T(n) = \Theta(\log n)$

Using The Master Theorem

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$ & $b > 1$

then

The Master Method Case 1

$T(n) = nT(n/3) + n$

where $a = 9$, $b = 3$, $f(n) = n$

& since $9^{(\log_3 n) - 1} = n^{\log_3 9 - 1} = n^{0.5}$

Thus the solution is $T(n) = \Theta(n^{0.5})$

Using The Master Method Case 2

$T(n) = 2T(n/3) + 1$

where $a = 2$, $b = 3$, $f(n) = 1$

& since $2^{(\log_3 n) - 1} = \Omega(1)$

Thus the solution is $T(n) = \Theta(\log n)$

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Mastr Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$$$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$$$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$$$

Note: The same results hold with $O$ instead of $\Theta$.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in \Theta(n^2)$

$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in \Theta(n^{2 \log n})$

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in \Theta(n^{3 \log n})$
Using The Master Method Case 3

\( T(n) = 3T(n/4) + n \log n \)

- \( a = 3, b = 4, f(n) = n \log n \)
- \( n^{\log_b a} = n^{\log_4 3} \approx 1.0989 \) (where \( c = 0.2 \))

**Case 3 applies:**

- Thus the solution is
  \( T(n) = \Theta(n^c \log n) \)

Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved.

Examples:
1. Computing \( a^n \) (\( a > 0 \), \( n \) a nonnegative integer)
2. Computing \( n! \)
3. Multiplying two matrices
4. Searching for a key of a given value in a list

Approaches to Algorithm Design

- Brute force
- Divide-and-Conquer
- Dynamic Programming
- Greedy method
- Backtracking
- Branch and Bound
- Local search
- Hill climbing, Simulated annealing
- Genetic algorithms, Genetic programming

Brute-Force Sorting Algorithm

**Selection Sort**
Scan the array to find its smallest element and swap it with the first element. Then, starting with the second element, scan the elements to the right of it to find the smallest among them and swap it with the second elements. Generally, on pass \( i \) (\( 0 \leq i \leq n-2 \)), find the smallest element in \( A[i..n-1] \) and swap it with \( A[i] \):

\[
A[0] \leq \ldots \leq A[i-1] \leq A[i] \leq \ldots \leq A[n-1]
\]

Example: 7 3 2 5

Analysis of Selection Sort

**Algorithm**

```
//Sorts a given array by selection sort
//Input: An array A[0..n-1] of n elements
//Output: Array A[0..n-1] sorted in ascending order
for i = 0 to n-2
do
    min = i
    for j = i+1 to n-1
do
        then
            min = j
    swap A[i] and A[min]
```

Time efficiency: \( \Theta(n^2) \)

Space efficiency: \( \Theta(1) \) on the place

Stability: \( \Theta(1) \)
Brute-Force String Matching

- **pattern**: a string of \( m \) characters to search for
- **text**: a (longer) string of \( n \) characters to search in
- **problem**: find a substring in the text that matches the pattern

**Brute-force algorithm**

1. **Step 1** Align pattern at beginning of text
2. **Step 2** Moving from left to right, compare each character of pattern to the corresponding character in text until
   - all characters are found to match (successful search); or
   - a mismatch is detected
3. **Step 3** While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Examples of Brute-Force String Matching

1. **Pattern**: 001011
   **Text**: 1001011010001011010

2. **Pattern**: happy
   **Text**: It is never too late to have a happy childhood.

Brute-Force Polynomial Evaluation

Problem: Find the value of polynomial

\[ p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \]

at a point \( x = x_0 \)

**Brute-force algorithm**

\[ p \leftarrow 0.0 \]

for \( i \leftarrow n \) down to 0 do

\[ \text{power} \leftarrow 1 \]

for \( j \leftarrow 1 \) to \( i \) do //compute \( x^j \)

\[ \text{power} \leftarrow \text{power} \times x \]

\[ p \leftarrow p + a[i] \times \text{power} \]

return \( p \)

**Efficiency**: \( \sum_{0 \leq i \leq n} i = \Theta(n^2) \) multiplications

Polynomial Evaluation: Improvement

We can do better by evaluating from right to left:

**Better brute-force algorithm**

\[ p \leftarrow a[0] \]

\[ \text{power} \leftarrow 1 \]

for \( i \leftarrow 1 \) to \( n \) do

\[ \text{power} \times \text{power} \times x \]

\[ p \leftarrow p + a[i] \times \text{power} \]

return \( p \)

**Efficiency**: \( \Theta(n) \) multiplications

Horner’s Rule is another linear time method.

Closest-Pair Problem

Find the two closest points in a set of \( n \) points (in the two-dimensional Cartesian plane).

**Brute-force algorithm**

Compute the distance between every pair of distinct points and return the indexes of the points for which the distance is the smallest.
Closest-Pair Brute-Force Algorithm (cont.)

**ALGORITHM** 

\[ \text{BruteForceClosestPoints}(P) \]

- **Input:** A list \( P \) of \( n \geq 2 \) points \( P_1 = (x_1, y_1), \ldots, P_n = (x_n, y_n) \)
- **Output:** Indices index1 and index2 of the closest pair of points

\[ d_{\text{min}} = \infty \]

for \( i \leftarrow 1 \) to \( n - 1 \) do
  for \( j \leftarrow i + 1 \) to \( n \) do
    \[ d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \] (\( \sqrt{q} \) is the square root function)
    if \( d < d_{\text{min}} \)
      then \( d_{\text{min}} \leftarrow d; \) index1 \( \leftarrow i; \) index2 \( \leftarrow j \)

return index1, index2

**Efficiency:** \( \Theta(n^2) \) multiplications (or \( \sqrt{q} \))

**How to make it faster?** Using divide and conquer

---

**Brute-Force Strengths and Weaknesses**

- **Strengths**
  - wide applicability
  - simplicity
  - yields reasonable algorithms for some important problems (e.g., matrix multiplication, sorting, searching, string matching)

- **Weaknesses**
  - rarely yields efficient algorithms
  - some brute-force algorithms are unacceptably slow
  - not as constructive as some other design techniques

---

**Exhaustive Search**

A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as permutations, combinations, or subsets of a set.

**Method:**
- generate a list of all potential solutions to the problem in a systematic manner (see algorithms in Sec. 5.4)
- evaluate potential solutions one by one, disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far
- when search ends, announce the solution(s) found

---

**Example 1: Traveling Salesman Problem**

- Given \( n \) cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively: Find shortest Hamiltonian circuit in a weighted connected graph

**Example:**

- Given 4 cities: a, b, c, d
- Distances:
  - ab: 2
  - ac: 8
  - ad: 5
  - bc: 3
  - bd: 7
  - cd: 4

**TSP by Exhaustive Search**

<table>
<thead>
<tr>
<th>Tour</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>a-b-c-d-a</td>
<td>2+3+7+5 = 17</td>
</tr>
<tr>
<td>a-b-d-c-a</td>
<td>2+4+7+8 = 21</td>
</tr>
<tr>
<td>a-c-b-d-a</td>
<td>8+3+4+5 = 20</td>
</tr>
<tr>
<td>a-c-d-b-a</td>
<td>8+7+4+2 = 21</td>
</tr>
<tr>
<td>a-d-b-c-a</td>
<td>5+4+3+8 = 20</td>
</tr>
<tr>
<td>a-d-c-b-a</td>
<td>5+7+3+2 = 17</td>
</tr>
</tbody>
</table>

**Efficiency:** \( \Theta(n!) \)

Chapter 5 discusses how to generate permutations fast.

---

**Example 2: Knapsack Problem**

Given \( n \) items:
- weights: \( w_1, w_2, \ldots, w_n \)
- values: \( v_1, v_2, \ldots, v_n \)
- a knapsack of capacity \( W \)

Find the most valuable subset of the items that fit into the knapsack.

**Example:** Knapsack capacity \( W = 16 \)

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>$20</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$30</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>$50</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>$10</td>
</tr>
</tbody>
</table>
Knapsack Problem by Exhaustive Search

### Example 3: The Assignment Problem

There are \( n \) people who need to be assigned to \( n \) jobs, one person per job. The cost of assigning person \( i \) to job \( j \) is \( C[i,j] \). Find an assignment that minimizes the total cost.

**Job 0**  |  **Job 1**  |  **Job 2**  |  **Job 3**
--- | --- | --- | ---
Person 0 | 9 | 2 | 7 | 8
Person 1 | 6 | 4 | 3 | 7
Person 2 | 5 | 8 | 1 | 8
Person 3 | 7 | 6 | 9 | 4

Algorithmic Plan: Generate all legitimate assignments, compute their costs, and select the cheapest one.

How many assignments are there? \( n! \)

Pose the problem as one about a cost matrix: cycle cover in a graph

Assignment Problem by Exhaustive Search

<table>
<thead>
<tr>
<th>Assignment (cols)</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>9 + 4 + 1 + 6 + 24 = 44</td>
</tr>
<tr>
<td>1, 2, 3, 5</td>
<td>9 + 4 + 6 + 9 + 30 = 54</td>
</tr>
<tr>
<td>1, 2, 3, 6</td>
<td>9 + 4 + 6 + 9 + 30 = 54</td>
</tr>
<tr>
<td>1, 2, 3, 7</td>
<td>9 + 4 + 6 + 9 + 30 = 54</td>
</tr>
<tr>
<td>1, 2, 3, 8</td>
<td>9 + 4 + 6 + 9 + 30 = 54</td>
</tr>
<tr>
<td>1, 2, 3, 9</td>
<td>9 + 4 + 6 + 9 + 30 = 54</td>
</tr>
<tr>
<td>1, 2, 3, 10</td>
<td>9 + 4 + 6 + 9 + 30 = 54</td>
</tr>
</tbody>
</table>

(Etc.)

(For this particular instance, the optimal assignment can be found by exploiting the specific features of the number given. It is: 1, 3, 4.)

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances.
- In some cases, there are much better alternatives!
  - Euler circuits
  - Shortest paths
  - Minimum spanning tree
  - Assignment problem (The Hungarian method runs in \( O(n^3) \) time.)
- In many cases, exhaustive search or its variation is the only known way to get exact solution.

Divide-and-Conquer

The most-well known algorithm design strategy:
1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions
Divide-and-Conquer Technique (cont.)

- A problem of size $n$
  - subproblem 1
    - of size $n/2$
    - a solution to subproblem 1
  - subproblem 2
    - of size $n/2$
    - a solution to subproblem 2
  - a solution to the original problem

General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d), \ d \geq 1$

Master Theorem:
- If $a < b^d$, $T(n) \in \Theta(n^d)$
- If $a = b^d$, $T(n) \in \Theta(n^d \log n)$
- If $a > b^d$, $T(n) \in \Theta(n^d \log \log n)$

Note: The same results hold with $O$ instead of $\Theta$.

Examples:
- $T(n) = 4T(n/2) + n \Rightarrow T(n) \in \Theta(n^2)$
- $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in \Theta(n^3 \log n)$
- $T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in \Theta(n^3)$

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort
- Binary tree traversals
- Binary search (T)
- Multiplication of large integers
- Matrix multiplication: Strassen’s algorithm
- Closest-pair and convex-hull algorithms

Pseudocode of Mergesort

```plaintext
// Sorts array A[0..n-1] by recursive mergesort
// Output: Array A[0..n-1] in non-decreasing order
Mergesort(A[0..n-1])
```

Pseudocode of Merge

```plaintext
// Merges two sorted arrays into one sorted array
// Outputs: Arrays B[0..p-1] and C[0..q-1] both sorted
// Input: Array A[0..p+q-1] of elements of B and C
Merge(A[0..p+q-1], B[0..p-1], C[0..q-1])
```
Mergesort Example

The non-recursive version of Mergesort starts from merging single elements into sorted pairs.

Analysis of Mergesort

- All cases have same efficiency: \( \Theta(n \log n) \)
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:
  \( \log_2 n! \approx n \log_2 n - 1.44n \)
- Space requirement: \( \Theta(n) \) (not in-place)
- Can be implemented without recursion (bottom-up)

Quicksort Example

- Selected a pivot (partitioning element) — here, the first element
- Rearrange the list so that all the elements in the first \( s \) positions are smaller than or equal to the pivot and all the elements in the remaining \( n-s \) positions are larger than or equal to the pivot (see next slide for an algorithm)
- Exchange the pivot with the last element in the first (i.e., \( s \)) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Analysis of Quicksort

- Best case: split in the middle — \( \Theta(n \log n) \)
- Worst case: sorted array! — \( \Theta(n^2) \)
- Average case: random arrays — \( \Theta(n \log n) \)
- Improvements:
  - better pivot selection: median of three partitioning
  - switch to insertion sort on small subfiles
  - elimination of recursion
  These combine to 20-25% improvement
- Considered the method of choice for internal sorting of large files (\( n \geq 10000 \))
**Design and Analysis of Algorithms**  
Chapter 3

### Binary Search

Very efficient algorithm for searching in sorted array:

\[ K \]

vs

\[ A[0] \ldots A[m] \ldots A[n-1] \]

If \( K = A[m] \), stop (successful search); otherwise, continue searching by the same method in \( A[0..m-1] \) if \( K < A[m] \) and in \( A[m+1..n-1] \) if \( K > A[m] \)

\[ \text{l} \leftarrow 0; \quad \text{r} \leftarrow \text{m}+1 \]

while \( \text{l} \leq \text{r} \) do

\[ m \leftarrow \lfloor (\text{l}+\text{r})/2 \rfloor \]

if \( K = A[m] \) return \( m \)

else if \( K < A[m] \) \( \text{r} \leftarrow m-1 \)

else \( \text{l} \leftarrow m+1 \)

return -1

Efficiency: \( \Theta(n) \). Why?

---

### Analysis of Binary Search

- Time efficiency:
  - worst-case recurrence: \( C_n = 1 + C_{\lceil n/2 \rceil} + C_{\lfloor n/2 \rfloor} \), \( C_1 = 1 \)
  - solution: \( C_n = \lceil \log(n+1) \rceil \)

  This is VERY fast: e.g., \( C_{10^6} = 20 \)

- Optimal for searching a sorted array
- Limitations: must be a sorted array (not linked list)
- Bad (degenerate) example of divide-and-conquer because only one of the sub-instances is solved
- Has a continuous counterpart called bisection method for solving equations in one unknown \( f(x) = 0 \) (see Sec. 12.4)

---

### Binary Tree Algorithms

Ex. 1: Classic traversals (preorder, inorder, postorder)

Algorithm Inorder(\( T \))

\[ \text{if } T \neq \emptyset \]

\[ \text{Inorder} (\text{left}(T)) \]

\[ \text{print(root of } T) \]

\[ \text{Inorder} (\text{right}(T)) \]

Efficiency: \( \Theta(n) \). Why? Each node is visited/printed once.

---

### Multiplication of Large Integers

Consider the problem of multiplying two (large) \( n \)-digit integers represented by arrays of their digits such as:

\[ A = 12345678901357986429 \quad B = 87654321284820912836 \]

The grade-school algorithm:

\[ \begin{align*}
& a_1 \cdot d_1 + b_2 \cdot d_2 + \ldots + b_n \cdot d_n \\
& (d_{20}) \cdot d_{19} \cdot \ldots \cdot d_1
\end{align*} \]

Efficiency: \( \Theta(n^2) \) single-digit multiplications

---

### First Divide-and-Conquer Algorithm

A small example: \( A \cdot B \) where \( A = 2135 \) and \( B = 4014 \)

\[ A = (21 \cdot 10^2 + 35), \quad B = (40 \cdot 10^2 + 14) \]

So, \( A \cdot B = (21 \cdot 10^4 + 35 \cdot 40 \cdot 10^2 + 14) \)

In general, if \( A = A_1 A_2 \) and \( B = B_1 B_2 \) (where \( A \) and \( B \) are \( n \)-digit, \( A_1, A_2, B_1, B_2 \) are \( n/2 \)-digit numbers),

\[ A \cdot B = A_1 \cdot B_1 \cdot 10^n + (A_1 \cdot B_2 + A_2 \cdot B_1) \cdot 10^n + A_2 \cdot B_2 \]

Recurrence for the number of one-digit multiplications \( M(n) \):

\[ M(n) = 4M(n/2), \quad M(1) = 1 \]

Solution: \( M(n) = n^2 \)
**Second Divide-and-Conquer Algorithm**

\[ A \times B = A_1 \times B_1 10^{10} + (A_1 \times B_2 + A_2 \times B_1) 10^{10} + A_2 \times B_2 \]

The idea is to decrease the number of multiplications from 4 to 3:

\[ (A_1 + A_2) \times (B_1 + B_2) = A_1 \times B_1 + (A_1 + B_2) \times A_2 + A_2 \times B_2 \]

Le., \((A_1 + A_2) \times (B_1 + B_2) = (A_1 \times B_1) + (B_1 \times A_2) - A_1 \times B_2 + A_2 \times B_1\), which requires only 3 multiplications at the expense of (4-1) extra add/sub.

**Recurrence for the number of multiplications** \(M(n)\):

\[ M(n) = 3M(n/2), M(1) = 1 \]

**Solution:** \(M(n) \approx \log_2 n \approx n^{1.585}\)

---

**Example of Large-Integer Multiplication**

\[ 2135 \times 4014 \]

\[ = (21 \times 10^2 + 35) \times (40 \times 10^2 + 14) \]

\[ = (21 \times 40^2) + (21 \times 14) + (35 \times 40) + (35 \times 14) \]

where \(c_1 = 2135 \times 40 + 21 \times 14 + 35 \times 40 + 35 \times 14\), and

\[ 21 \times 40 = (2 \times 40 + 1) \times (10 \times 40 + 1) \]

\[ = (2 \times 40^2) + (2 \times 40) + 1 \]

where \(c_2 = (21 \times 40) + 2 \times 40^2 + 1\).

This process requires 9 digit multiplications as opposed to 16.

---

**Conventional Matrix Multiplication**

\[
\begin{array}{cc}
A_{10} & A_{11} \\
C_{00} & C_{01} \\
C_{10} & C_{11}
\end{array} = \begin{array}{cc}
A_{00} & a_{01} \\
A_{10} & a_{11}
\end{array} \times \begin{array}{cc}
b_{00} & b_{01} \\
b_{10} & b_{11}
\end{array}
\]

\[= a_{00} \times b_{00} + a_{01} \times b_{01} + a_{10} \times b_{10} + a_{11} \times b_{11}\]

\[= 8 \text{ multiplications} \quad \text{Efficiency class in general: } \Theta(n^2)\]

---

**Formulas for Strassen’s Algorithm**

\[M_1 = (A_{10} + A_{11}) \times (B_{00} + B_{11}) \]

\[M_2 = (A_{10} + A_{11}) \times B_{00} \]

\[M_3 = A_{00} \times B_{01} + B_{11} \]

\[M_4 = (A_{00} + A_{10}) \times (B_{00} + B_{10}) \]

\[M_5 = (A_{00} + A_{10}) \times B_{11} \]

\[M_6 = (A_{10} - A_{00}) \times (B_{00} + B_{10}) \]

\[M_7 = (A_{10} - A_{00}) \times B_{11} \]

\[= 7 \text{ multiplications} \quad \text{18 additions}\]
Analysis of Strassen’s Algorithm

If \( n \) is not a power of 2, matrices can be padded with zeros.

Number of multiplications:

\[
M(n) = 7M(n/2), \quad M(1) = 1
\]

Solution: \( M(n) = 7 \log_2 n = n \log_2 7 \approx n^{2.807} \)

Algorithms with better asymptotic efficiency are known but they are even more complex and not used in practice.

Closest Pair by Divide-and-Conquer

Step 0 Sort the points by x (list one) and then by y (list two).

Step 1 Divide the points given into two subsets \( S_1 \) and \( S_2 \) by a vertical line \( x = c \) so that half the points lie to the left or on the line and half the points lie to the right or on the line.

Step 2 Find recursively the closest pairs for the left and right subsets.

Step 3 Set \( d = \min|d_1, d_2| \)

We can limit our attention to the points in the symmetric vertical strip of width \( 2d \) as possible closest pair. Let \( C_1 \) and \( C_2 \) be the subsets of points in the left subset \( S_1 \) and of the right subset \( S_2 \), respectively, that lie in this vertical strip. The points in \( C_1 \) and \( C_2 \) are stored in increasing order of their y coordinates, taken from the second list.

Step 4 For every point \( P(x,y) \) in \( C_1 \), we inspect points in \( C_2 \) that may be closer to \( P \) than \( d \). There can be no more than \( 6 \) such points (because \( d \leq d_2 \)).

Efficiency of the Closest-Pair Algorithm

Running time of the algorithm (without sorting) is:

\[
T(n) = 2T(n/2) + M(n), \quad \text{where } M(n) \in \Theta(n)
\]

By the Master Theorem (with \( a = 2, b = 2, d = 1 \))

\[
T(n) = \Theta(n \log n)
\]

So the total time is \( \Theta(n \log n) \).

Convex hull: smallest convex set that includes given points. An \( O(n^3) \) brute force time is given in Levitin, Ch 3.

Assume points are sorted by x-coordinate values

Identify extreme points \( P_1 \) and \( P_2 \) (leftmost and rightmost)

Compute upper hull recursively:

1. Find point \( P_{\text{max}} \) that is farthest away from line \( P_1P_2 \)
2. Compute the upper hull of the points to the left of line \( P_1P_{\text{max}} \)
3. Compute the upper hull of the points to the left of line \( P_{\text{max}}P_2 \)

Compute lower hull in a similar manner
Efficiency of Quickhull Algorithm

- Finding point farthest away from line \(P_1P_2\) can be done in linear time.
- Time efficiency:
  \[ T(n) = T(x) + T(y) + T(z) + T(v) + O(n), \]
  where \(x + y + z + v \leq n\).
- worst case: \(\Theta(n^2)\)
- average case: \(\Theta(n)\) (under reasonable assumptions about distribution of points given)

- If points are not initially sorted by \(x\)-coordinate value, this can be accomplished in \(O(n \log n)\) time.

- Several \(O(n \log n)\) algorithms for convex hull are known.

Example: Fibonacci numbers

- Recall definition of Fibonacci numbers:
  \[ F(n) = F(n-1) + F(n-2) \]
  \[ F(0) = 0 \]
  \[ F(1) = 1 \]

- Computing the \(n\)th Fibonacci number recursively (top-down):
  \[ F(n) \]
  \[ F(n-1) \]
  \[ F(n-2) \]
  \[ F(n-3) \]
  \[ F(n-4) \]

- Efficiency:
  - time: \(n\)
  - space: \(n\)
  What if we solve it recursively?

Dynamic Programming

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- “Programming” here means “planning”
- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Examples of DP algorithms

- Computing a binomial coefficient
- Longest common subsequence
- Warshall’s algorithm for transitive closure
- Floyd's algorithm for all-pairs shortest paths
- Constructing an optimal binary search tree
- Some instances of difficult discrete optimization problems:
  - traveling salesman
  - knapsack

Dynamic Programming is a general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances.
Let $\textit{Value of } V = \textit{optimal value of such an instance.}$ Then

\[V(i,j) = \begin{cases} \max\{V(i-1,j), V(i-1,j-w_i)\} & \text{if } w_i \leq j \\
V(i-1,j) & \text{if } w_i > j
\end{cases}\]

Initial conditions: $V(0,j) = 0$ and $V(i,0) = 0$.

Knapsack Problem by DP (pseudocode)

Algorithm DPPKnapsack($I[..n], V[..n,..W])$
var $V[0...n,0...W]$, $P[0...n,0...W]$: int
for $i := 0$ to $W$ do $V[0,i] := 0$
for $i := 0$ to $n$ do $V[i,0] := 0$
for $i := 1$ to $W$ do
for $j := 1$ to $n$ do
if $w_i \leq j$ and $V[i-1,j] + V[i-1,j-w_i] > V[i,j]$ then
$V[i,j] := V[i-1,j] + V[i-1,j-w_i]$, $P[i,j] := j-w_i$
else
return $V[n,W]$ and the optimal subset by backtracing

Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

<table>
<thead>
<tr>
<th>item</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>$12$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$10$</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$20$</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$15$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>capacity</th>
<th>item</th>
<th>weight</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>

Knapsack finds the actual optimal subset, i.e. solution.
How to compute LCS?

1. Let $A = a_1a_2...a_n$ and $B = b_1b_2...b_m$.
2. $\text{lens}(i, j)$ the length of an LCS between $a_1a_2...a_i$ and $b_1b_2...b_j$.
3. With proper initializations, $\text{lens}(i, j)$ can be computed as follows.

$$
\text{lens}(i, j) = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0; \\
\text{lens}(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j; \\
\max(\text{lens}(i-1, j), \text{lens}(i, j-1)) & \text{if } i, j > 0 \text{ and } a_i \neq b_j.
\end{cases}
$$

The backtracing algorithm

```plaintext
procedure LCS-Length(A, B)
1. for $i = 0$ to $n$ do $\text{lens}(i, 0) = 0$
2. for $j = 0$ to $m$ do $\text{lens}(0, j) = 0$
3. for $i = 1$ to $n$ do
   for $j = 1$ to $m$ do
      if $a_i = b_j$ then
         $\text{lens}(i, j) = \text{lens}(i-1, j-1) + 1$
      else
         $\text{lens}(i, j) = \max(\text{lens}(i-1, j), \text{lens}(i, j-1))$
5. if $\text{lens}(n, m) > 0$ then return $\text{lens}(n, m)$.
6. else return 0.
```

Running time and memory: $O(nm)$ and $O(nm)$. 

Another example:

Sequence 1: algorithm
Sequence 2: alignment
One of its LCS is algm.

Procedure LCS-Length(A, B):
1. for $i = 0$ to $n$ do $\text{lens}(i, 0) = 0$
2. for $j = 0$ to $m$ do $\text{lens}(0, j) = 0$
3. for $i = 1$ to $n$ do
   for $j = 1$ to $m$ do
      if $a_i = b_j$ then
         $\text{lens}(i, j) = \text{lens}(i-1, j-1) + 1$
      else
         $\text{lens}(i, j) = \max(\text{lens}(i-1, j), \text{lens}(i, j-1))$
5. if $\text{lens}(n, m) > 0$ then return $\text{lens}(n, m)$.
6. else return 0.

The backtracing algorithm

```plaintext
procedure Output-LCS(A, prev, i, j)
1. if $i = 0$ or $j = 0$ then return
2. if $\text{prev}(i, j) = \uparrow$ then $\text{Output-LCS}(A, \text{prev}, i-1, j)$
3. else if $\text{prev}(i, j) = \downarrow$ then $\text{Output-LCS}(A, \text{prev}, i, j-1)$
4. else $\text{Output-LCS}(A, \text{prev}, i-1, j-1)$
```

Copyright © 2007 Pearson Addison-Wesley. All rights reserved.
Computes the transitive closure of a relation

\[
\begin{array}{cccc|cccc}
 i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Output: priden

---

**Warshall’s Algorithm (matrix generation)**

**Recurrence relating elements \( R^k \) to elements of \( R^{k+1} \):**

\[
R^k[i,j] = R^{k-1}[i,j] \text{ or } R^{k-1}[i,k] \text{ and } R^{k-1}[k,j]
\]

It implies the following rules for generating \( R^k \) from \( R^{k-1} \):

**Rule 1** If an element in row \( i \) and column \( j \) is 1 in \( R^{k-1} \), it remains 1 in \( R^k \).

**Rule 2** If an element in row \( i \) and column \( j \) is 0 in \( R^{k-1} \), it has to be changed to 1 in \( R^k \) if and only if the element in its row \( i \) and column \( k \) and the element in its column \( j \) and row \( k \) are both 1’s in \( R^{k-1} \).

**Warshall’s Algorithm (example)**

\[
R^{(0)} = \begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

\[
R^{(1)} = \begin{bmatrix}
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
\]

\[
R^{(2)} = \begin{bmatrix}
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1
\end{bmatrix}
\]

\[
R^{(3)} = \begin{bmatrix}
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1
\end{bmatrix}
\]

**Constructs transitive closure \( T \) as the last matrix in the sequence of \( n \times n \) matrices \( R^{(0)}, \ldots, R^{(k)}, \ldots, R^{(n)} \) where \( R^{(k)}[i,j] = 1 \) iff there is nontrivial path from \( i \) to \( j \) with only the first \( k \) vertices allowed as intermediate.**

Note that \( R^{(0)} = A \) (adjacency matrix), \( R^{(n)} = \overline{T} \) (transitive closure).
Warshall’s Algorithm (pseudocode and analysis)

ALGORITHM Warshall(A[1..n, 1..n])
//Implements Warshall’s algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph

R[0] = A  // initialize R with A

for k = 1 to n  // for each vertex
    for i = 1 to n  // for each vertex
        for j = 1 to n  // for each vertex
            R[k][i][j] = min(R[k-1][i][j], R[k-1][i][k] + R[k-1][k][j])

return R[n]

Time efficiency: \( O(n^3) \)
Space efficiency: Matrices can be written over their predecessors (with some care), so it’s \( O(n^2) \).

Floyd’s Algorithm (example)

Problem: Given a weighted (di)graph, find shortest paths between every pair of vertices.

Same idea: construct solution through series of matrices \( D^{(k)}, \ldots, D^{(n)} \) using increasing subsets of the vertices allowed as intermediate.

Example:

\[
\begin{array}{cccc}
  & 1 & 2 & 3 \\
1 & 0 & 4 & \infty \\
2 & 1 & 0 & \infty \\
3 & \infty & \infty & 0 \\
\end{array}
\]

\[
D^{(0)} = \begin{bmatrix}
0 & \infty & 3 & \infty \\
2 & 0 & \infty & \infty \\
5 & 7 & 0 & 1 \\
6 & \infty & 9 & 0 \\
\end{bmatrix}
\]

\[
D^{(1)} = \begin{bmatrix}
0 & \infty & 3 & \infty \\
2 & 0 & 5 & \infty \\
5 & 7 & 0 & 1 \\
6 & \infty & 9 & 0 \\
\end{bmatrix}
\]

\[
D^{(2)} = \begin{bmatrix}
0 & \infty & 3 & 4 \\
2 & 0 & 5 & 6 \\
9 & 7 & 0 & 1 \\
6 & \infty & 9 & 0 \\
\end{bmatrix}
\]

Optimal Binary Search Trees

Problem: Given \( n \) keys \( a_1 < \cdots < a_n \), and probabilities \( p_1, \ldots, p_n \), searching for them, find a BST with a minimum average number of comparisons in successful search.

Since total number of BSTs with \( n \) nodes is given by \( C(2n,n)/(n+1) \), which grows exponentially, brute force is hopeless.

Example: What is an optimal BST for keys \( A, B, C \), and \( D \) with search probabilities \( 0.1, 0.2, 0.4 \), and 0.3, respectively?

Average # of comparisons

\[
1 \cdot 0.4 + 2 \cdot 0.2 + 0.3 + 3 \cdot 0.1 = 1.7
\]
Design and Analysis of Algorithms

Chapter 3

DP for Optimal BST Problem

Let \( C(i,j) \) be minimum average number of comparisons made in \( T(i,j) \), optimal BST for keys \( a_i < \ldots < a_j \), where \( 1 \leq i \leq j \leq n \).

Consider optimal BST among all BSTs with some \( a_k (i \leq k \leq j) \) as their root; \( T(i,j) \) is the best among them.

Optimal BST for

\[
\begin{align*}
C(i,j) &= \min_{1 \leq k \leq j} \left[ p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot \text{level}(a_s \text{ in } T(i,k-1) +1) + \sum_{s=k+1}^{j} p_s \cdot \text{level}(a_s \text{ in } T(k+1,j) +1) \right] + s + 1
\end{align*}
\]

Optimal Binary Search Trees

Algorithm (OptimalBST(\(P \), \(L \))):

1. Finds an optimal binary search tree by dynamic programming.
2. Input: An array \( P \) of search probabilities for a sorted list of \( n \) keys.
3. Output: Average number of comparisons in successful searches in the
   optimal BST and table \( R \) of subtrees' roots in the optimal BST.

\[
\begin{align*}
&\text{for } i = 1 \text{ to } n \text{ do} \\
&\quad \text{ } C(i, i-1) = 0 \\
&\quad \text{ } N(i, i) = i \\
&\quad \text{ } C(i, i) = 0 \\
&\text{for } d = 1 \text{ to } n-1 \text{ do} \text{ (diagonal count) } \\
&\quad \text{ } \text{for } i = 1 \text{ to } n-d \text{ do} \\
&\quad \quad \text{ } j = i + d \\
&\quad \quad \text{ } \text{minval} \leftarrow \infty \\
&\quad \quad \text{ } \text{for } k = i \text{ to } j \text{ do} \\
&\quad \quad \quad \text{ } \text{if } \{ 1, i-1 \} \cup \{ k+1, j \} \text{ is minimal} \\
&\quad \quad \quad \quad \text{ } \text{minval} \leftarrow \text{in } \{ 1, i-1 \} \cup \{ k+1, j \} \\
&\quad \quad \quad \quad \text{ } \text{root} \leftarrow k \\
&\quad \quad \quad \quad \text{ } \text{sum} \leftarrow p[k] \\
&\quad \quad \quad \quad \text{ } \text{for } s = i \text{ to } j \text{ do} \text{ inc } \text{sum} \leftarrow \text{sum} + p[s] \\
&\quad \quad \quad \quad \text{ } \text{return } C(i,j), R
\end{align*}
\]

Analysis DP for Optimal BST Problem

Time efficiency: \( \Theta(n^3) \) but can be reduced to \( \Theta(n^2) \) by taking advantage of monotonicity of entries in the root table, i.e., \( R(i,j) \) is always in the range between \( R(i-1,j) \) and \( R(i+1,j) \).

Space efficiency: \( \Theta(n^2) \)

Method can be expanded to include unsuccessful searches

Example: key

\[
\begin{array}{cccc}
A & B & C & D \\
\text{probability} & 0.1 & 0.2 & 0.4 & 0.3
\end{array}
\]

The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

\[
C(i,j) = \min_{1 \leq k \leq j} \left[ C(i,k-1) + C(k+1,j) + \sum_{s=i}^{k-1} p_s \cdot \text{level}(a_s \text{ in } T(i,k-1) +1) + \sum_{s=k+1}^{j} p_s \cdot \text{level}(a_s \text{ in } T(k+1,j) +1) \right]
\]

the right one, for trees' roots, records \( k \) values giving the minima

Greedy Technique

optimal BST
Greedy Technique

Constructs a solution to an optimization problem piece by piece through a sequence of choices that are:

1) feasible, i.e. satisfying the constraints
2) locally optimal (with respect to some neighborhood definition)
3) greedy (in terms of some measure), and irrevocable

For some problems, it yields a globally optimal solution for every instance. For most, does not but can be useful for fast approximations. We are mostly interested in the former case in this class.

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

- Approximations/heuristics:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Minimum Spanning Tree (MST)

- Spanning tree of a connected graph G: a connected acyclic subgraph of G that includes all of G’s vertices
- Minimum spanning tree of a weighted, connected graph G: a spanning tree of G of the minimum total weight

Example:

Prim’s MST algorithm

- Start with tree T₁ consisting of one (any) vertex and “grow” tree one vertex at a time to produce MST through a series of expanding subtrees T₁, T₂, ..., Tₙ
- On each iteration, construct Tᵢ₊₁ from Tᵢ by adding vertex not in Tᵢ that is closest to those already in Tᵢ (this is a “greedy” step!)
- Stop when all vertices are included

Change-Making Problem

Given unlimited amounts of coins of denominations \(d₁ > \ldots > dₘ\), give change for amount \(n\) with the least number of coins

- What are the objective function and constraints?

Example: \(d₁ = 25c\), \(d₂ = 10c\), \(d₃ = 5c\), \(d₄ = 1c\), and \(n = 48c\)

Greedy solution: \(<1, 2, 0, 3>\)

Greedy solution is

- optimal for any amount and “normal” set of denominations
- not optimal for arbitrary coin denominations
Notes about Prim’s algorithm

- Proof by induction that this construction actually yields an MST (CLRS, Ch. 23.1). Main property is given in the next page.
- Needs priority queue for locating closest fringe vertex. The detailed algorithm can be found in Levitin, P. 310.
- Efficiency
  - $O(n^2)$ for weight matrix representation of graph and array implementation of priority queue
  - $O(m \log n)$ for adjacency lists representation of graph with $n$ vertices and $m$ edges and min-heap implementation of the priority queue

The Crucial Property behind Prim’s Algorithm

Claim: Let $G = (V,E)$ be a weighted graph and $(X,Y)$ be a partition of $V$ (called a cut). Suppose $e = (x,y)$ is an edge of $E$ across the cut, where $x$ is in $X$ and $y$ is in $Y$, and $e$ has the minimum weight among all such crossing edges (called a light edge). Then there is an MST containing $e$.

Notes about Kruskal’s algorithm

- Algorithm looks easier than Prim’s but is harder to implement (checking for cycles!)
- Cycle checking: a cycle is created iff added edge connects vertices in the same connected component
- Union-find algorithms – see section 9.2
- Runs in $O(m \log m)$ time, with $m = |E|$. The time is mostly spent on sorting.

Another greedy algorithm for MST: Kruskal’s

- Sort the edges in nondecreasing order of lengths
- “Grow” tree one edge at a time to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

Minimum spanning tree vs. Steiner tree

In general, a Steiner minimal tree (SMT) can be much shorter than a minimum spanning tree (MST), but SMTs are hard to compute.
Shortest paths – Dijkstra’s algorithm

**Single Source Shortest Paths Problem:** Given a weighted connected directed graph G, find shortest paths from source vertex s to each of the other vertices.

**Dijkstra’s algorithm:** Similar to Prim’s MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex u with the smallest sum

\[ d_u + w(v,u) \]

where

- v is a vertex for which shortest path has been already found
- \( d_u \) is the length of the shortest path from source s to u
- \( w(v,u) \) is the length (weight) of edge from v to u

**Notes on Dijkstra’s algorithm**

1. Correctness can be proven by induction on the number of vertices.
2. Doesn’t work for graphs with negative weights (whereas Floyd’s algorithm does, as long as there is no negative cycle). Can you find a graph for Dijkstra’s algorithm?
3. Applicable to both undirected and directed graphs
4. Efficiency
   - \( O(V^2) \) for graphs represented by weight matrix and array implementation of priority queue
   - \( O((E+V)\log V) \) for graphs represented by adj. lists and min-heap implementation of priority queue
5. Don’t mix up Dijkstra’s algorithm with Prim’s algorithm! More details of the algorithm are in the text and web books.

Coding Problem

**Coding:** assignment of bit strings to alphabet characters

E.g. We can code \{a,b,c,d\} as \{00,01,10,11\} or \{00,01,01,10\} or \{00,01,00,10\}.

**Codewords:** bit strings assigned for characters of alphabet.

**Two types of codes:**
- **fixed-length encoding** (e.g., ASCII)
- **variable-length encoding** (e.g., Morse code)

E.g. If \( P(a) = 0.4, P(b) = 0.3, P(c) = 0.2, P(d) = 0.1 \), then the average length of code 82 is \( 4 \times 0.4 + 2 \times 0.3 + 3 \times 0.2 + 3 \times 0.1 = 1.9 \) bits.

**Prefix-free codes (or prefix-codes):** no codeword is a prefix of another codeword

It allows for efficient (online) decoding! E.g. consider the encoded string (msg) 10010110...

**Problem:** If frequencies of the character occurrences are known, what is the best binary prefix-free code?

The one with the shortest average code length. The average code length represents on the average how many bits are required to transmit or store a character.

Prefix-codes (e.g., ASCII)

Huffman codes

- Any binary tree with edges labeled with 0’s and 1’s yields a prefix-free code of characters assigned to its leaves.
- Optimal binary tree minimizing the average length of a codeword can be constructed as follows:

**Huffman’s algorithm**

Initialize n one-node trees with alphabet characters and the tree weights with their frequencies.

Repeat the following step n-1 times: join two binary trees with smallest weights into one (as left and right subtrees) and make its weight equal the sum of the weights of the two trees.

Mark edges leading to left and right subtrees with 0’s and 1’s, respectively.

Example

<table>
<thead>
<tr>
<th>Tree vertices</th>
<th>Remaining vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(-,0)</td>
<td>b(-,3)</td>
</tr>
<tr>
<td>b(a,3)</td>
<td>c(b,3+4)</td>
</tr>
<tr>
<td>d(b,5)</td>
<td>c(b,7)</td>
</tr>
<tr>
<td>c(b,7)</td>
<td>e(d,9)</td>
</tr>
</tbody>
</table>

E.g. if \( P(a) = 0.4, P(b) = 0.3, P(c) = 0.2, P(d) = 0.1 \), then the average length of code 82 is \( 4 \times 0.4 + 2 \times 0.3 + 3 \times 0.2 + 3 \times 0.1 = 1.9 \) bits.

Notes on Dijkstra’s algorithm

1. Correctness can be proven by induction on the number of vertices.
2. Does’t work for graphs with negative weights (whereas Floyd’s algorithm does, as long as there is no negative cycle). Can you find a graph for Dijkstra’s algorithm?
3. Applicable to both undirected and directed graphs
4. Efficiency
   - \( O(V^2) \) for graphs represented by weight matrix and array implementation of priority queue
   - \( O((E+V)\log V) \) for graphs represented by adj. lists and min-heap implementation of priority queue
5. Don’t mix up Dijkstra’s algorithm with Prim’s algorithm! More details of the algorithm are in the text and web books.

Example

<table>
<thead>
<tr>
<th>Character</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>frequency</td>
<td>0.35</td>
<td>0.6</td>
<td>0.2</td>
<td>0.15</td>
</tr>
<tr>
<td>codeword</td>
<td>11 100 00 01 101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>average bits per character</td>
<td>2.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for fixed-length encoding</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>compression ratio</td>
<td>(3-2.25)/3*100% = 25%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Design and Analysis of Algorithms  

Chapter 3

Backtracking & Branch and Bound
Busby, Dodge, Fleming, and Negrusa

Example: N-Queens Problem

Given an N x N sized chess board
Objective: Place N queens on the board so that no queens are in danger

Backtracking Algorithm

- Is used to solve problems for which a sequence of objects is to be selected from a set such that the sequence satisfies some constraint
- Traverses the state space using a depth-first search with pruning

Backtracking

- Performs a depth-first traversal of a tree
- Continues until it reaches a node that is non-viable or non-promising
- Prunes the sub tree rooted at this node and continues the depth-first traversal of the tree

Backtracking

- Backtracking prunes entire sub trees if their root node is not a viable solution
- The algorithm will “backtrack” up the tree to search for other possible solutions

Example: N-Queens Problem

One option would be to generate a tree of every possible board layout
This would be an expensive way to find a solution

Example: N-Queens Problem

One option would be to generate a tree of every possible board layout
This would be an expensive way to find a solution
Efficiency of Backtracking

- This gives a significant advantage over an exhaustive search of the tree for the average problem.
- The worst case: The algorithm tries every path, traversing the entire search space as in an exhaustive search.

Example: The Traveling Salesman Problem

- Branch and bound can be used to solve the TSP using a priority queue as an auxiliary data structure.
- An example is the problem with a directed graph given by this adjacency matrix:

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>4</td>
<td>10</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>7</td>
<td>8</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>7</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>7</td>
<td>17</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
```

Branch and Bound

- Where backtracking uses a depth-first search with pruning, the branch and bound algorithm uses a breadth-first search with pruning.
- Branch and bound uses a queue as an auxiliary data structure.

Traveling Salesman Problem

- The problem starts at vertex 1.
- The initial bound for the minimum tour is the sum of the minimum outgoing edges from each vertex:

```
Vertex 1: min(14, 4, 10, 20) = 4
Vertex 2: min(14, 7, 8, 7) = 7
Vertex 3: min(4, 5, 7, 16) = 4
Vertex 4: min(11, 7, 9, 2) = 2
Vertex 5: min(18, 7, 17, 4) = 4
Bound = 21
```

The Branch and Bound Algorithm

- Starting by considering the root node and applying a lower-bounding and upper-bounding procedure to it.
- If the bounds match, then an optimal solution has been found and the algorithm is finished.
- If they do not match, then algorithm runs on the child nodes.

Traveling Salesman Problem

- Next, the bound for the node for the partial tour from 1 to 2 is calculated using the formula:

\[ \text{Bound} = \text{Length from } 1 \text{ to } 2 + \text{sum of min outgoing edges for vertices from 2 to 5} = 14 + (7 + 4 + 2 + 4) = 31 \]
Traveling Salesman Problem

- The node is added to the priority queue
- The node with the lowest bound is then removed
- This calculation for the bound for the node of the partial tours is repeated on this node
- The process ends when the priority queue is empty

The final results of this example are in this tree:

The accompanying number for each node is the order it was removed in

Efficiency of Branch and Bound

- In many types of problems, branch and bound is faster than branching, due to the use of a breadth-first search instead of a depth-first search
- The worst case scenario is the same, as it will still visit every node in the tree

Local Search

Approach quite general used in:
- planning, scheduling, routing, configuration, protein folding, etc
- In fact, many (most?) large Operations Research problems are solved using local search methods (e.g., airline scheduling).
Local Search

Key idea (surprisingly simple):
- Select (random) initial state (initial guess at solution)
- Make local modification to improve current state
- Repeat Step 2 until goal state found (or out of time)

Requirements:
- generate an initial (often random; probably-not-optimal or even valid) guess
- evaluate quality of guess
- move to other states (well-defined neighborhood function)

... and do these operations quickly
... and don’t save paths followed

Hill-climbing search

\[ \text{"Like climbing Everest in thick fog with amnesia"} \]

\[ \rightarrow \text{Move to a better "neighbor"} \]

function Hill-Climbing(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
neighbor, a node

current = MAKE-INITIAL-STATE(problem)
while neighbor succeeds current do
    neighbor = a highest-valued successor of current
    if \text{VALUE}(neighbor) \leq \text{VALUE}(current) then return \text{STATE}(current)
    current = neighbor

Note: "successor" normally called neighbor

Local search for CSPs

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned
- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values
- Variable selection: randomly select any conflicted variable
- Value selection by min-conflicts heuristic:
  - choose value that violates the fewest constraints
  - i.e., hill-climb with \text{h(n)} = total number of violated constraints

Queens

- States: 4 queens in 4 columns (256 states)
- Neighborhood Operators: move queen in column
- Goal test: no attacks
- Evaluation: \text{h(n)} = number of attacks

Remark

- Local search with min-conflict heuristic works extremely well for n-queen problems
- The reason: Solutions are densely distributed in the \(O(n^2)\) space, which means that on the average a solution is a few steps away from a randomly picked assignment
Stochastic Hill-climbing search: 8-queens problem

Picking next move **randomly** from among uphill moves.

- $h = 17$ for the above state
- Operators: move queen in column - best move leads to $h=12$ (marked with square)

Stochastic hill climbing:

- Pick next move **randomly** from among uphill moves.
- $h_i$ for the above state
- Operators: move queen in column - best move leads to $h=12$ (marked with square)

Problems with Hill Climbing

- Foot hills / Local Optimal: No neighbor is better, but not at global optimum.
  - (Maze: may have to move AWAY from goal to find (best) solution)
- Plateaus: All neighbors look the same.
  - (8-puzzle: perhaps no action will change # of tiles out of place)
- Ridges: sequence of local maxima
- Ignorance of the peak: Am I done?

Hill-climbing search: 8-queens problem

- What’s the value of the evaluation function $h$, (num. of attacking queens)?
- $h = 3$
- Find a move to improve it.

Local beam search

- Start with $k$ randomly generated states
- Keep track of $k$ states rather than just one
- At each iteration, all the successors of all $k$ states are generated
- If any one is a goal state, stop; else select the $k$ best successors from the complete list and repeat.
- Equivalent to $k$ random-restart hill-climbing???

No: This is different than $k$ random restarts since information is shared between $k$ search points:

Some search points may contribute none to best successors: one search point may contribute all $k$ successors “Come over here, the grass is greener” (Russell and Norvig)
Design and Analysis of Algorithms  Chapter 3

Improvements to Basic Local Search

- Issue:
  - How to move more quickly to successively better plateaus?
  - Avoid “getting stuck” / local maxima?
- Idea: Introduce downhill moves (“noise”) to escape from plateaus/local maxima
- Noise strategies:
  1. Simulated Annealing
     - Kirkpatrick et al. 1982; Metropolis et al. 1953
  2. Mixed Random Walk (Satisfiability)
     - Selman and Kautz 1993

Simulated Annealing

- Idea:
  Use conventional hill-climbing style techniques, but occasionally take a step in a direction other than that in which there is improvement (downhill moves).
  As time passes, the probability that a down-hill step is taken is gradually reduced and the size of any down-hill step taken is decreased.
  - Kirkpatrick et al. 1982; Metropolis et al. 1953

Notes on Simulated Annealing

- Noise model based on statistical mechanics
  - Introduced as analogue to physical process of growing crystals
  - Kirkpatrick et al. 1982; Metropolis et al. 1953
- Convergence:
  - 1. W/ exponential schedule, will converge to global optimum
  - 2. No more-precise convergence rate (Recent work on rapidly mixing Markov chains)
- Key aspect: downwards / sideways moves
  - Expensive, but if have enough time can be best
  - Hundreds of papers/ year
    - Many applications: VLSI layout, factory scheduling, protein folding...

Genetic Algorithms

- Another class of iterative improvement algorithms
  - A genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators
  - Inspired by the biological evolution process
  - Uses concepts of “Natural Selection” and “Genetic Inheritance” (Darwin 1859)
  - Originally developed by John Holland (1975)

High-level Algorithm

- Idea: escape local maxima by allowing some “bad” moves but gradually decrease their frequency

  Similar to hill climbing, but a random chance of improvement makes the moves

  What is the probability when: \( T \to 0 \)?
  - The probability where: \( A = 0 \)
  - The probability when: \( T \to \infty \)

  Otherwise, move the state with probability the improvement over the current state.

  Genetic Algorithms

  1. Randomly generate an initial population.
  2. Evaluate the fitness population.
  3. Select parents and “reproduce” the next generation.
  4. Replace the old generation with the new generation.
  5. Repeat step 2 through 4 till iteration N
Stochastic Operators

- **Cross-over**
  - decomposes two distinct solutions and then
  - randomly assigns their parts to form novel solutions

- **Mutation**
  - randomly perturbs a candidate solution

---

Genetic algorithms

- **A successor state is generated by combining two parent states**
- **Start with k randomly generated states (population)**
- **A state is represented as a string over a finite alphabet**
  (often a string of 0s and 1s)
- **Evaluation function (Fitness function). Higher values for better states.**
- **Produce the next generation of states by selection, crossover, and mutation**

- **Lots of variants of genetic algorithms with different selection, crossover, and mutation rules.**
- **GA have a wide application in optimization – e.g., circuit layout and job shop scheduling**
- **Much work remains to be done to formally understand GA’s and to identify the conditions under which they perform well.**
Summary

- Local search algorithms
  - Hill-climbing search
  - Local beam search
  - Simulated annealing search
  - Genetic algorithms

Solving Recurrence Relations

So what does
\( T(n) = T(n-1) + n \)
look like anyway?

Kurt Schmidt
Drexel University

Local Search Summary

- Surprisingly efficient search technique
- Wide range of applications
- Formal properties elusive
- Intuitive explanation:
  - Search spaces are too large for systematic search anyway...
- Area will most likely continue to thrive

Recurrence Relations

- Can easily describe the runtime of recursive algorithms
- Can then be expressed in a closed form (not defined in terms of itself)
- Consider the linear search:

Eg. 1 - Linear Search

Recursively
Look at an element (constant work, \( c \)), then search the remaining elements...

- \( T(n) = T(n-1) + c \)
- “The cost of searching \( n \) elements is the cost of looking at 1 element, plus the cost of searching \( n-1 \) elements”

The End
Linear Search (cont.)

Caveat:
- You need to convince yourself (and others) that the single step, examining an element, "is" done in constant time.
- Can I get to the i-th element in constant time, either directly, or from the (i-1)th element?
- Look at the code

Methods of Solving Recurrence Relations

- Substitution (we'll work on this one in this lecture)
- Accounting method
- Draw the recursion tree, think about it
- The Master Theorem*
- Guess at an upper bound, prove it

* See Cormen, Leiserson, & Rivest, Introduction to Algorithms

Looking for Patterns

- Note, the intermediate results are enumerated
- We need to pull out patterns, to write a general expression for the k-th unwinding
  - This requires practice. It is a little bit art. The brain learns patterns, over time. Practice.
- Be careful while simplifying after substitution

Linear Search (cont.)

- We'll "unwind" a few of these
  \[ T(n) = T(n-1) + c \]  
  But, \( T(n-1) = T(n-2) + c \), from above
  Substituting back in:
  \[ T(n) = T(n-2) + c + c \]
  Gathering like terms
  \[ T(n) = T(n-2) + 2c \]  

Eg. 1 – list of intermediates

<table>
<thead>
<tr>
<th>Result at i-th unwinding</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = T(n-1) + 1 )</td>
<td>1</td>
</tr>
<tr>
<td>( T(n) = T(n-2) + 2 )</td>
<td>2</td>
</tr>
<tr>
<td>( T(n) = T(n-3) + 3 )</td>
<td>3</td>
</tr>
<tr>
<td>( T(n) = T(n-4) + 4 )</td>
<td>4</td>
</tr>
</tbody>
</table>
Design and Analysis of Algorithms  Chapter 3

Linear Search (cont.)

- An expression for the kth unwinding:
  \[ T(n) = T(n-k) + kc \]
- We have 2 variables, k and n, but we have a relation
- \( T(d) \) is constant (can be determined) for some constant d (we know the algorithm)
- Choose any convenient # to stop.

Let's decide to stop at \( T(0) \). When the list to search is empty, you're done...

Let \( n-k = 0 \) => \( n=k \)

Now, substitute n in everywhere for k:

\[ T(n) = T(n-n) + nc \]
\[ T(n) = T(0) + nc = nc + c_0 = O(n) \] (\( T(0) \) is some constant, \( c_0 \))

Binary Search (cont.)

Let's do some quick substitutions:

\[ T(n) = T(n/2) + c \] (1)

where \( c \) is some constant, the cost of checking the middle...

We need to write an expression for the kth unwinding (in n & k)

- Must find patterns, changes, as \( k = 1, 2, ..., k \)
- This can be the hard part
- Do not get discouraged! Try something else...
- We'll re-write those equations...
- We will then need to relate n and k

Algorithm – “check middle, then search lower \( \frac{1}{2} \) or upper \( \frac{1}{2} \)"

Let's do some quick substitutions:

\[ T(n) = T(n/2) + c \] (1)

\[ T(n/2) = T(n/4) + c \]

\[ T(n) = T(n/4) + 2c \] (2)

\[ T(n/4) = T(n/8) + c \]

\[ T(n) = T(n/8) + 2c \] (3)

Result at i

- \( i = 1 \) => \( T(n/2) + c \)
- \( i = 2 \) => \( T(n/4) + 2c \)
- \( i = 3 \) => \( T(n/8) + 3c \)
- \( i = 4 \) => \( T(n/16) + 4c \)
Binary Search (cont.)

Result at \( i^{th} \) unwinding

\[
\begin{align*}
T(n) &= T(n/2) + c \\
T(n) &= T(n/4) + 2c \\
T(n) &= T(n/8) + 3c \\
T(n) &= T(n/16) + 4c
\end{align*}
\]

\( i \)

1
2
3
4

\[T(n) = T(n/2^k) + kc\]

After \( k \) unw windings:

Need a convenient place to stop unwinding – need to relate \( k \) & \( n \)

Let’s pick \( T(0) = c_0 \). So,

\[
\frac{n}{2^k} = 0 \Rightarrow n=0
\]

Hmm. Easy, but not real useful...

**Substituting back in (getting rid of k):**

\[
T(n) = T(1) + c \log(n)
\]

\[
= c \log(n) + c_0
\]

\( = O(\log(n)) \)

Okay, let’s consider \( T(1) = c_0 \)

So, let:

\[
\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_n n
\]