CSOR W4231 Midterm Notes

0. Basic knowledge

0.1 Definition

1. An algorithm is efficient if it achieves better worst case performance than **brute-force** search (暴力解法)
2. An algorithm is efficient if it has a polynomial running time (多项式运行时间)
3. Divide and conquer: break problems into several subproblems, then solve each subproblem independently, finally combine the solutions of the subproblems to solve the original problem

0.2 Correctness

A formal proof often by induction

> 归纳（induction）是一种证明数学命题的方法，它通过对于所有自然数的特定断言的证明来得出一个命题的正确性。具体来说，归纳分为数学归纳法和强归纳法两种形式。
>
> 数学归纳法的基本思想是：证明当 n = 1 时命题成立，然后假设当 n = k 时命题成立，证明当 n = k+1 时命题也成立。因此，该方法需要两个步骤：基础步骤和归纳步骤。基础步骤是证明当 n = 1 时命题成立。归纳步骤是假设当 n = k 时命题成立，然后证明当 n = k+1 时命题也成立。因此，当我们证明了基础步骤和归纳步骤，就证明了该命题对于所有自然数都成立。
>
> 强归纳法与数学归纳法类似，不同之处在于归纳步骤假设所有小于 k+1 的数都满足命题，然后证明当 n = k+1 时命题也成立。强归纳法通常用于证明涉及比较复杂的数学对象的命题，例如树和图等。

Example:

0.3 Running time

Number of primitive computational steps (原始的计算步骤)

- 并不是指特定的运行时间，在CS中我们需要将硬件所带来的对运行时间的影响消除。

0.4 Asymptotic Notation

- Upper bound: Big O
- Lower bound: Big Ω
- Tight bound: Big Θ

Rules:

- 忽略n前面的数字(10n -> n)
- 在底数相同时，首选幂次大的(n^5)，舍弃幂次小的(n^3)
- 首选指数函数(2^n)，舍弃多项式函数(n^3)

- 首选多项式(n)，舍弃log函数($log^3(n)$)

$$\log n < n < n\log n < n^2 < 2^n < 3^n < n^n$$

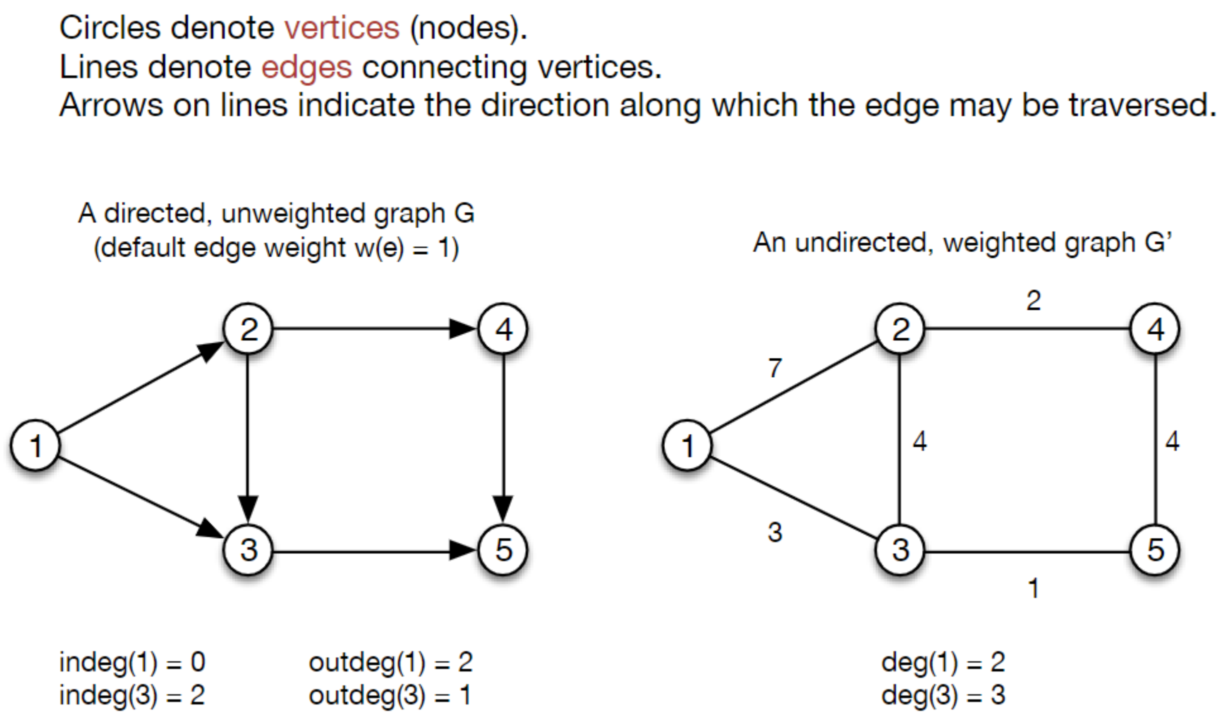
0.5 Mater Theorem

If $T(n) = aT(\lceil n/b \rceil) + \text{O}(n^k)$ for some constants $a > 0, b > 1, k \geq 0$, then

$$T(n) = \begin{cases} \text{O}(n^{log_b a}), & if\ a > b^k \\ \text{O}(n^k logn), & if\ a = b^k \\ \text{O}(n^k), & if\ a < b^k \end{cases}$$

0.6 Graph

- 有向图(directed graph)包含了有限的节点(node)V和E组有向边(directed edge)。有向边是一组节点的对(u, v)。在数学表示中，有向图G被表示为$G = (V, E)$。有向图常用于表示网页链接，社交关系等。
- 无向图(undirected graph)的边没有方向性。无向图通常用于地图中的道路，原子之间的连接等。



Circles denote vertices (nodes).
Lines denote edges connecting vertices.
Arrows on lines indicate the direction along which the edge may be traversed.

A directed, unweighted graph G
(default edge weight w(e) = 1)

An undirected, weighted graph G'

indeg(1) = 0
indeg(3) = 2

outdeg(1) = 2
outdeg(3) = 1

deg(1) = 2
deg(3) = 3

- (1, 2, 3, 2, 4) in G' is a **path**
- **A path is simple** when all nodes are distinct
- (1, 2, 3, 1) in G' is a **simple cycle**
- G' is **connected**
- The **connected component** of 1 in G' is {1, 2, 3, 4, 5}
- **Strongly connected component**: node u has a path from u to v and v to u.
  E.g., node 1 has strongly connected component in G is 1
- **Bipartite graphs:** nodes can be split into two subnets such that there are no edges between nodes in the same subnet

0.7 NP-complete problem

1.**P类问题**：存在多项式时间算法的问题。(P：polynominal，多项式)。

2.**NP问题**：能在多项式时间内验证得出一个正确解的问题。(NP:Nondeterministic polynominal，非确定性多项式)。

NP完备是一个计算机科学中的概念，用于描述一类计算问题的复杂性。

NP（Nondeterministic Polynomial）是指可以在多项式时间内验证解的一类计算问题。具体来说，对于NP问题，可以在多项式时间内验证一个解是否正确，但无法在多项式时间内找到一个正确的解。

NP完备（NP-complete）则是指一类NP问题，这些问题是NP中最难的一类问题。NP完备问题的特点是，如果能在多项式时间内解决其中任何一个问题，那么就能在多项式时间内解决NP中的所有问题。也就是说，NP完备问题是NP中最难的问题，目前没有已知的多项式时间算法能够解决它们。

NP完备问题包括许多著名的计算问题，例如旅行商问题、背包问题、子集和问题等等。这些问题在实际应用中都具有重要的意义，但由于它们的计算复杂性非常高，通常需要使用近似算法或启发式算法来求解。

由于NP完备问题的计算复杂性非常高，它们对于计算机科学的研究和发展具有重要的意义。解决NP完备问题的算法或理论可能会引发革命性的计算机科学进展。

0.8 Dynamic Programming

动态规划是通过拆分问题，定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决。这里的递推可以理解为一种递归的反向。

递推不是关键，关键是如何拆分问题，这就涉及到了对于问题状态的定义和状态转移方程的定义。
动态规划的本质，是对问题状态的定义和状态转移方程的定义。

状态的定义：是对问题的重新描述，使得问题在不同的时刻，有不同状态。
状态转移方程：是状态和状态之间转换的关系式，一般为递归表达式的方式。此时应当明确递归的初始状态，即边界条件。

1. Optimal substructure: the optimal solution to the problem contains optimal solutions to the subproblems.
2. A recurrence for the overall optimal solution in terms of optimal solutions to appropriate subproblems. The recurrence should provide a natural ordering of the subproblems from smaller to larger and require polynomial work for combining solutions to the subproblems.
3. Iterative, bottom-up computation of subproblems, from smaller to larger.
4. Small number of subproblems (polynomial in n).

DP vs. Divide & Conquer

- They both combine solutions to subproblems to generate the overall solution.
- However, divide and conquer starts with a large problem and divides it into small pieces.
- While dynamic programming works from the bottom up, solving the smallest subproblems first and building optimal solutions to steadily larger problems.

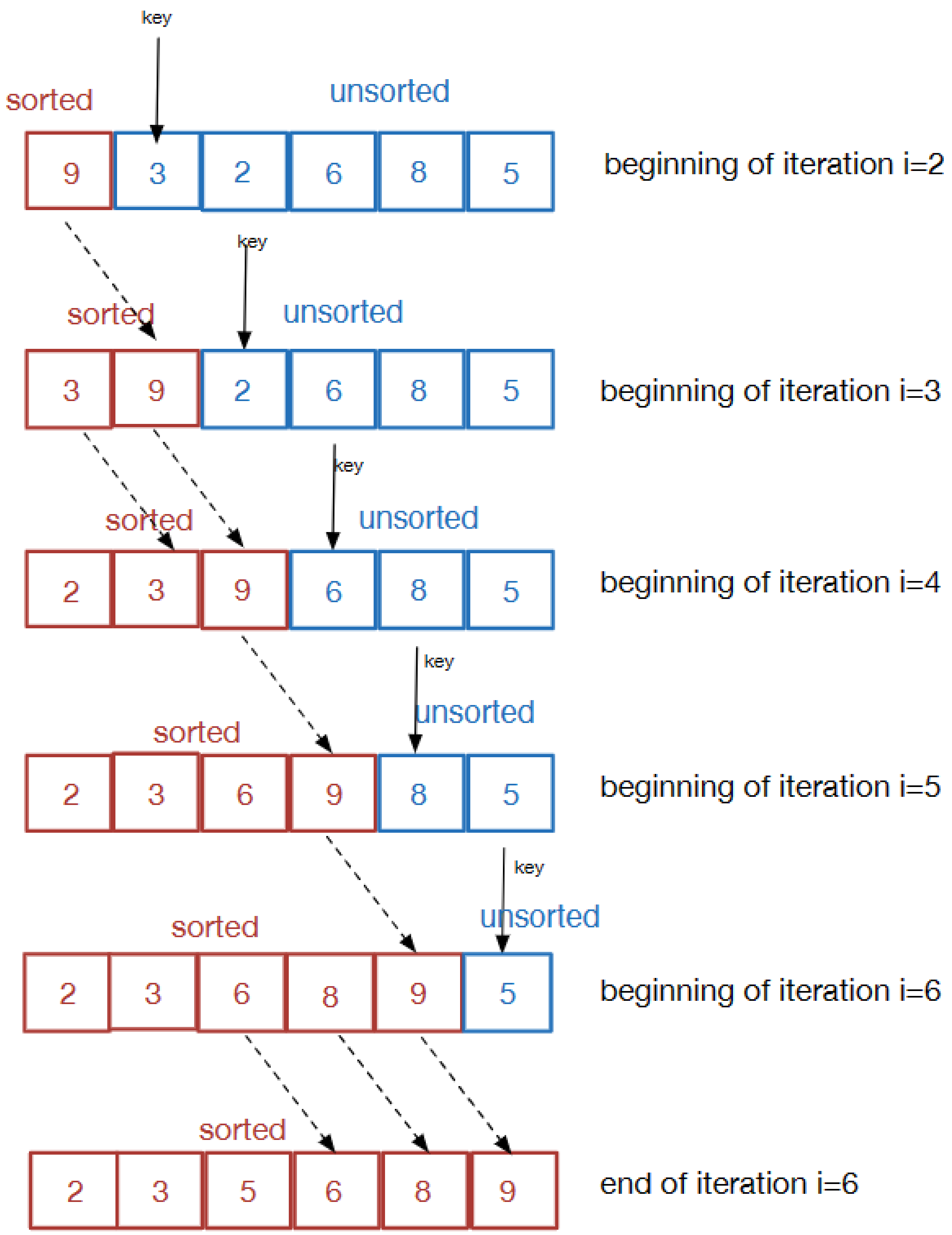
1. Insertion sort

1.1 Main idea

1. 从一个有序序列开始，即将第一个元素看作一个有序序列(sorted subarray)，称作为数组A[1]，其余的被称为未排序序列(unsorted subarray)
2. 将此有序序列的大小增加1，方法是将下一个元素A(key)，插入到有序数组的适当位置。
   - 将key和每个有序序列中的元素x进行比较，从最右边的数字开始。
     - 如果$x > key$，将x向右移一格
     - 如果$x \leq key$，将key插入在x的后面
3. 重复第二步，直到未排序序列为0

1.2 Example

$n = 6, A = \{9, 3, 2, 6, 8, 5\}$



1.3 Pseudo Code

```
insertion-sort(A)
    for i=2 to n do
        key = A[i]
        // Insert A[i] into the sorted subarray A[1,i-1]
        j = i - 1
        while j>0 and A[j]>key do
            A[j+1] = A[j]
            j = j - 1
        end while
        A[j+1] = key
    end for
```

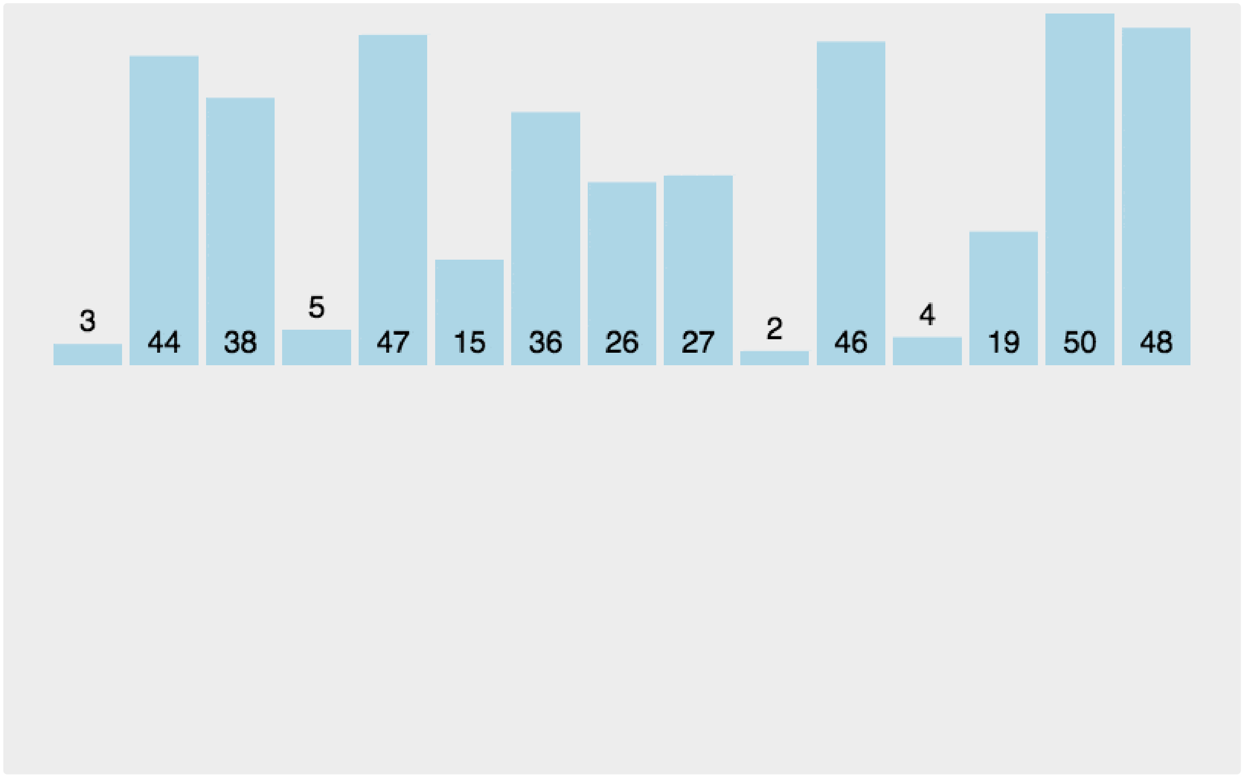
1.4 Running Time

Each elements needs to be shifted to the beginning of the array, takes $O(n)$ time. And there are total n elements, so the running time should be $O(n^2)$

2. Merge Sort (Divide & Conquer)

2.1 Main idea

1. 建立两个数组L,R，将待排序数组以中间分开，分别将左右两边的数字放入array
2. 设置两个指针，分别指向两个数组的第一个位置
3. 比较两个指针指向的元素，选择**较小**的一个元素移动到合并空间，并移动该指针到下一个位置
4. 重复步骤3直到某一指针到达末尾
5. 将指针未到末尾的数组中的数字直接复制到合并空间队尾



2.2 Pseudo Code

```
merge(A, left, right, mid)
    L = A[left, mid]
    R = A[mid+1, right]
    Maintain two pointers p_l, p_r, initialised to point to the first elements of L,R, repectively
    while both lists are non-empty do
        let x,y be the elements pointed to by p_l, p_r
        compare x,y and append the smaller to the output
        advance the pointer in the list with the smaller of x,y
    end while
    append the remainder of the non-empty list to the output
mergesort(A, left, right)
    if right == left then return
    end if
    mid = left + lower_bound((right-left)/2) //下取整(right-left)/2
    mergesort(A, left, mid)
    mergesort(A, mid+1, right)
    merge(A, left, right, mid)
```

2.3 Running Time

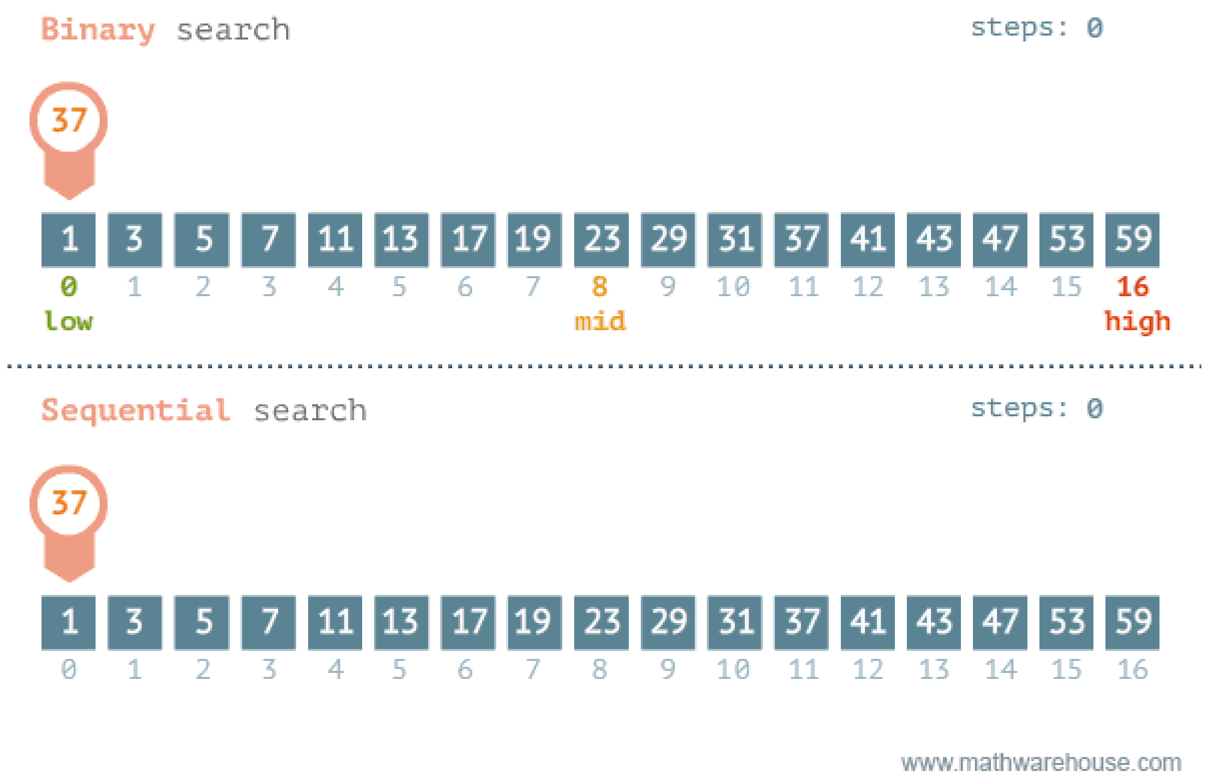
- L R 分别有$n/2$个元素
- 在最坏情况下，循环中将会遍历$n-1$个元素
- Running time will be: $\mathrm{O}(n)$

3. Binary Search

3.1 Main Idea

- 找到特定的数字

1. 从数组的中间位元素开始
2. 如果需要找到的数字比中间位小，则在小于中间位的那一半继续寻找，反之亦然
3. 重复步骤2，直到找到数字



3.2 Pseudo Code

```
binarysearch(A, left, right)
    mid = left + d(right -left)/2e
    if x == A[mid] then
        return mid
    else if right == left then
        return no
    else if x > A[mid] then
        left = mid + 1
    else right = mid -1
    end if
```

3.3 Running time

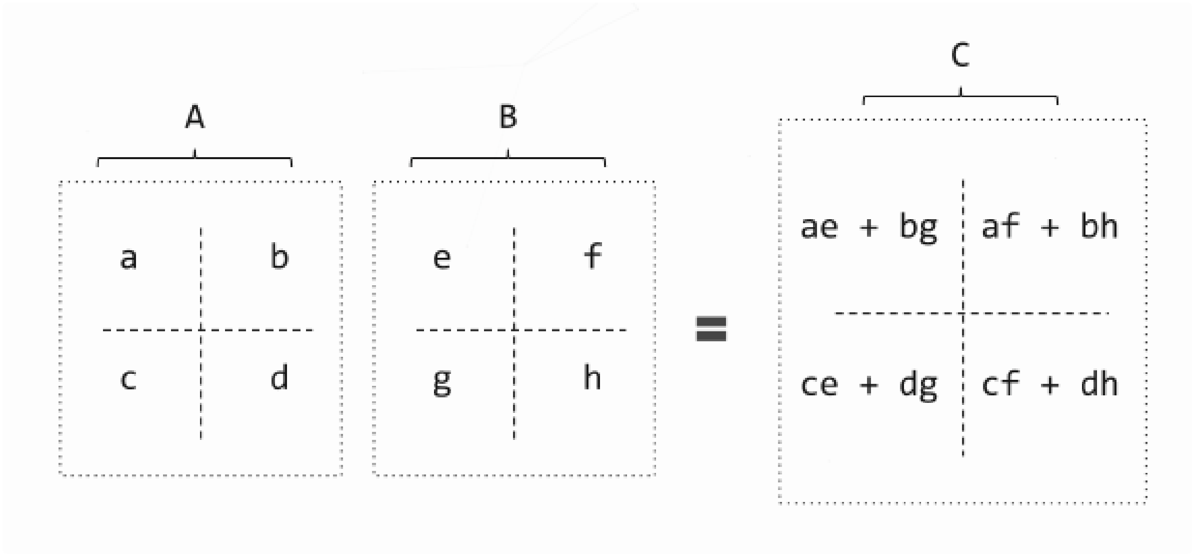
- 由于每次二分舍弃的是$n/2$的数据量，所以需要处理的数据量是$n/2$.即$T(n) \le T(n/2) + \mathrm{O}(1)$
- 根据Master Theorem，可以得出running time：$T(n) = \mathrm{O}(\log n)$

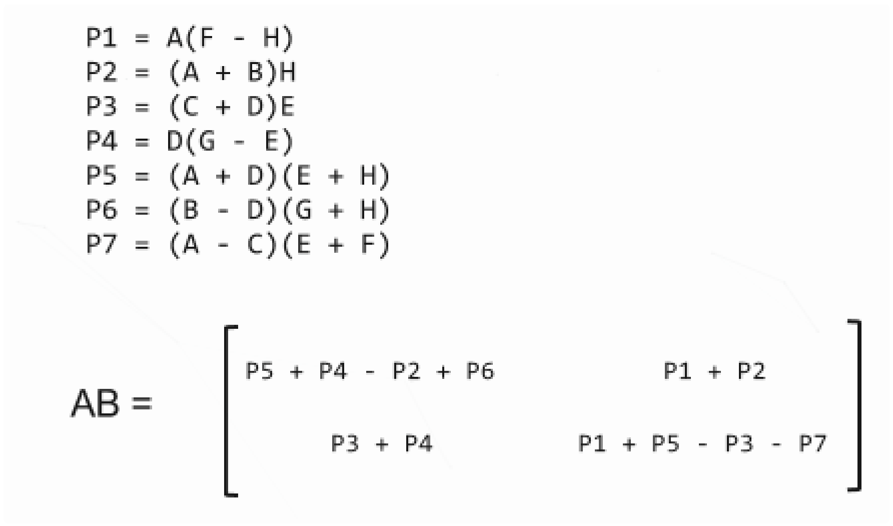
4. Strassen's algorithm

- Strassen's algorithm是一种计算矩阵乘法的算法，暴力解法使用了三个循环，其复杂度为$\mathrm{O}(n^3)$，非常大。
- 由于我们知道，矩阵乘法中包含乘法和加法。乘法的复杂度较大

4.1 Main Idea

1. 将矩阵乘法拆分成7个子问题，定义7个变量：





```
P1 = A(F - H)
P2 = (A + B)H
P3 = (C + D)E
P4 = D(G - E)
P5 = (A + D)(E + H)
P6 = (B - D)(G + H)
P7 = (A - C)(E + F)
```

$$AB = \begin{bmatrix} P5 + P4 - P2 + P6 & P1 + P2 \\ P3 + P4 & P1 + P5 - P3 - P7 \end{bmatrix}$$

2. 用这7个变量去计算乘法。

4.2 Running Time

Recurrence: $T(n) = 7T(n/2) + cn^2$

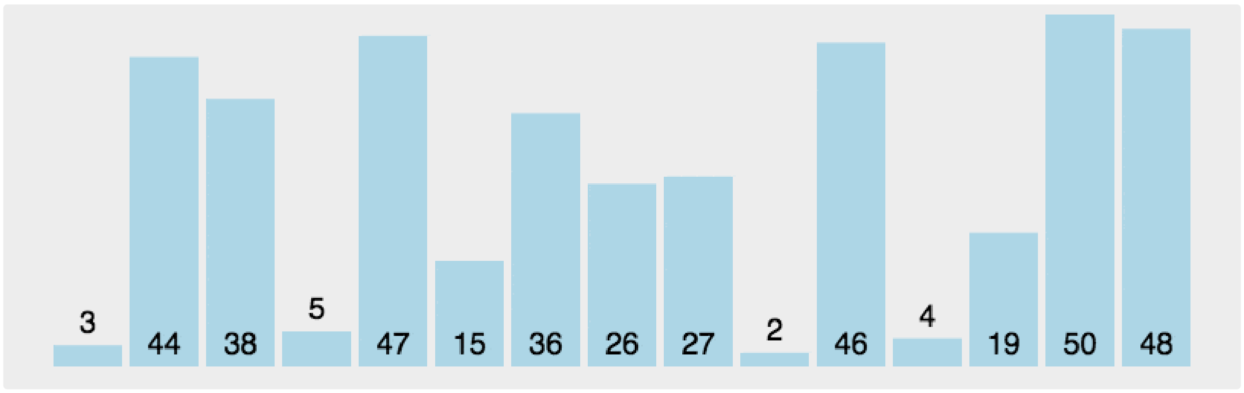
By the Master Theorem: $T(n) = O(n^{log_2 7}) = O(n^{2.81})$

5. Quicksort (Divide & Conquer)

- Divide and conquer algorithm
- In-place algorithm
- Worst case running time is $\Theta(n^2)$, but the average case running time is $\Theta(n \log n)$

5.1 Main idea

1. 从数列中挑出一个元素，称为 "基准"（pivot）；
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；



5.2 Pseudo Code

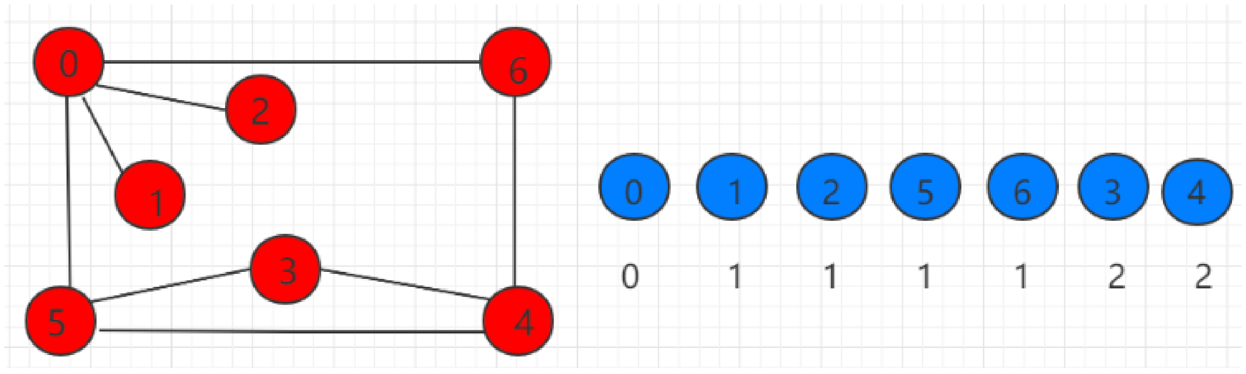
```
Quicksort(A,left,right)
    if |A|= 0 then return //A is empty
    end if
    split = Partition(A,left,right)
    Quicksort(A,left,split -1)
    Quicksort(A,split + 1,right)
```

5.3 Running Time

- 在best case中，假设每次分区时，基准是input的中位数：
  $T(n) = 2T(n/2) + \Theta(n) = \mathrm{O}(n\log n)$
  这也是average case
- Worst case：每个数都要处理
  $T(n) = \mathrm{O}(n^2)$

6. Breadth First Search (BFS)

- 广度优先算法(BFS)是按照层数(layer)搜索的.
- 广度优先遍历从某个顶点 v 出发，首先访问这个结点，并将其标记为已访问过，然后顺序访问结点v的所有未被访问的邻接点 {vi,..,vj} ，并将其标记为已访问过，然后将 {vi,...,vj} 中的每一个节点重复节点v的访问方法，直到所有结点都被访问完为止。



6.1 Main Idea

- 使用一个辅助队列 q，首先将顶点 v 入队，将其标记为已访问，然后循环检测队列是否为空。
- 如果队列不为空，则取出队列第一个元素，并将与该元素相关联的所有未被访问的节点入队，将这些节点标记为已访问。
- 如果队列为空，则说明已经按照广度优先遍历了所有的节点。

6.2 Pseudo Code

```
BFS(G = (V, E), s ∈V )
    array discovered[V] initialized to 0
    array dist[V] initialized to ∞
    array parent[V] initialized to N IL
    queue q
    discovered[s] = 1
    dist[s] = 0
    parent[s] = NULL
    enqueue(q, s)
    while size(q) > 0 do
        u =dequeue(q)
        for (u, v) ∈ E do
            if discovered[v] == 0 then
                discovered[v] = 1
                dist[v] = dist[u] + 1
                parent[v] = u
                enqueue(q, v)
            end if
        end for
    end while
```

6.3 Running Time

For the BFS algorithm, the running time is $\mathrm{O}(n+E)$, where n is the number of nodes, equivalent to V, and E is the number of edges, since the worst cast for BFS is that the algorithm needs to discover all edges and visit all nodes.

6.4 2-colourability

- If a graph contains an **odd-length cycle**, then it is **not** 2-colourable

7. Depth First Search (DFS)

- Starting from a node s, explore the graph as deeply as possible, then backtrack
- 深度优先遍历(Depth First Search)的主要思想是首先以一个未被访问过的顶点作为起始顶点，沿当前顶点的边走到未访问过的顶点。当没有未访问过的顶点时，则回到上一个顶点，继续试探别的顶点，直至所有的顶点都被访问过。

7.1 Main Idea

1. Try the first edge out of s, towards some node v
2. Continue from v until reach a **dead end**, that is a node whose neighbors have all been explored
3. **Backtrack** to the first node with an unexplored neighbor and repeat step 2

7.2 Pseudo Code

```
DFS(G = (V, E))
    for u ∈V do
        explored[u] = 0
    end for
    for u ∈V do
        if explored[u] == 0 then Search(u)
        end if
    end for
Search(u)
    previsit(u)
    explored[u] = 1
    for (u, v) ∈E do
        if explored[v] == 0 then Search(v)
        end if
    end for
    postvisit(u)
```

7.3 Running Time

For the DFS algorithm, the running time is $\mathrm{O}(n+E)$, where n is the number of nodes, equivalent to V, and E is the number of edges, since the worst cast for BFS is that the algorithm needs to discover all edges and visit all nodes.

7.4 Strongly Connected Components (SCC)

8. Dijkstra’s algorithm

- **Greedy principle:** a local decision rule is applied at every step
- Dijkstra’s algorithm is **greedy**: always from the shortest new $s-v$ path by first following a path to some node u in S, and then a single edge (u,v)

8.1 Main Idea

Dijkstra算法是一种用于求解单源最短路径问题的贪心算法。它通过逐步扩展已处理节点的方式计算每个节点到起点的最短路径，具体步骤如下：

1. 初始化
   将起点的距离设为0，其他节点的距离设为无穷大。同时，将起点加入未处理节点集合中，其他节点加入已处理节点集合中。
2. 迭代扩展节点
   从未处理节点集合中选择距离起点最近的节点，将其加入已处理节点集合中。然后，对于该节点的每个邻居节点，执行松弛操作，即更新邻居节点的距离值（如果通过当前节点到达邻居节点的距离比之前计算的距离更短，就更新邻居节点的距离值）。
3. 重复执行步骤2，直到到达终点或者未处理节点集合为空为止。
4. 输出最短路径
   根据已处理节点集合中每个节点到起点的距离值，可以得到每个节点到起点的最短路径。

8.2 Pseudo Code

```
Dijkstra-v3(G = (V, E, w), s ∈V )
    Initialize(G, s)
    Q = BuildQueue({V ; dist})
    S = Ø
    while Q 6= Ø do
        u = ExtractMin(Q)
        S = S U{u}
        for (u, v) ∈E do
            Update(u, v)
        end for
    end while
```

8.3 Running Time

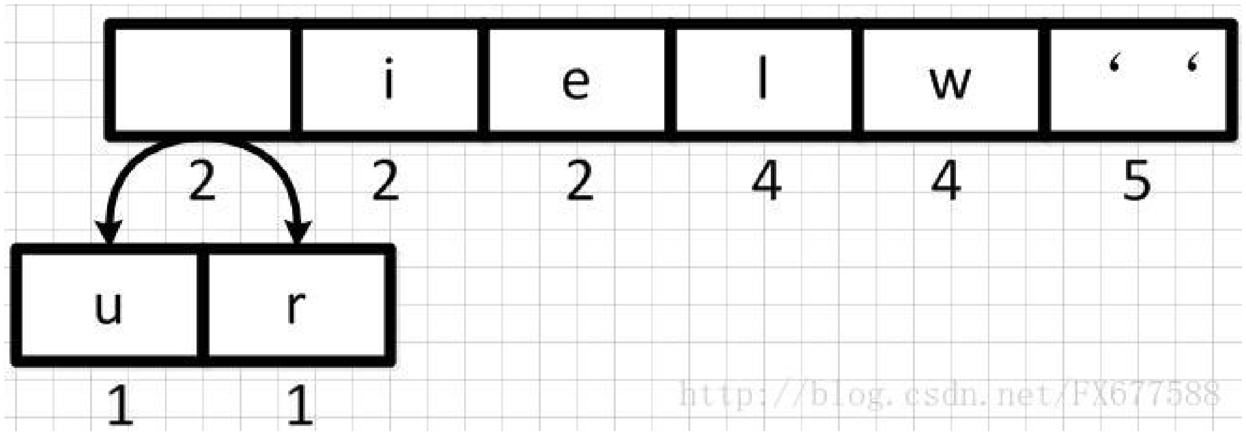
$$T(n) = O(n\log n + m\log n) = O(m\log n),\ m = V + E,\ n = V$$

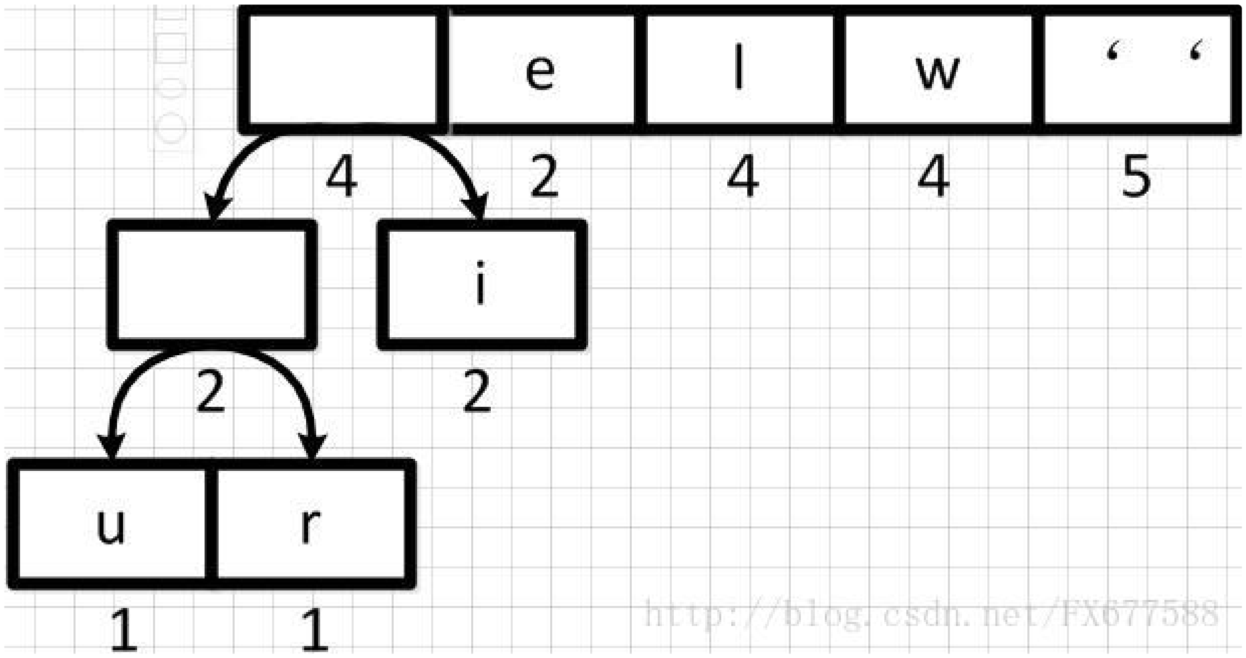
9. Huffman Algorithm

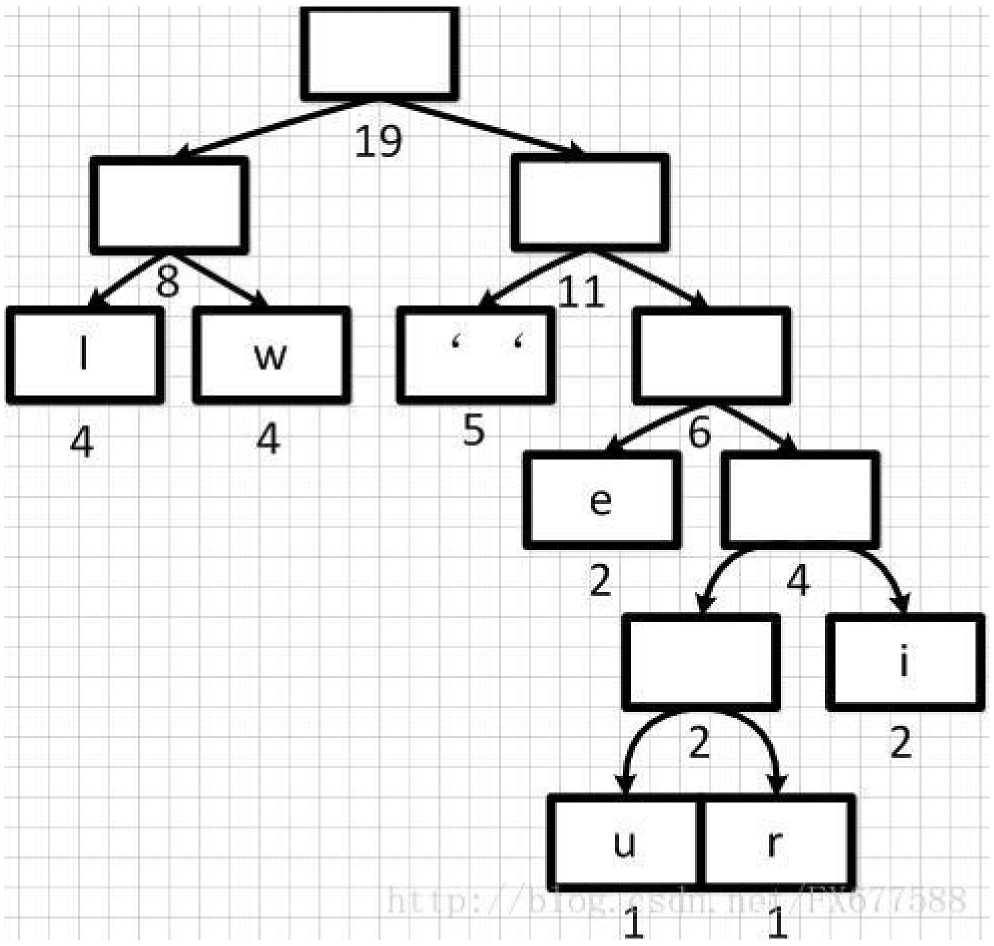
- 压缩图片，文本，音频的主要思路是：将使用频率最高的几个字符统计出来，使用一种占用空间更小的命名系统重新命名。
- 在执行时，需要注意：
    - 主流的字符命名系统，如ASCII使用7 bit，是能够保证一串字符中没有任意一个字符当作另一个字符的前缀，即不会误判为其他字符

9.1 Main Idea

- Huffman算法使用二叉树将使用的字符按照频率排序，并将他们放入二叉树中，使用频率越高的字符在更浅的地方，占用bit越少
- 二叉树的每个分支最多有两个子节点
- 二叉树的左侧都由0命名，右侧为1







9.2 Pseudo Code

```
Huffman(A, P)
    if |A|= 2 then
        Encode one symbol using 0 and the other using 1
    end if
    Let α and β be the two symbols with the lowest probabilities
    Let v be a new meta-character with probability Pr[α] + Pr[β]
    Let A1 = A-{α,β}+ {v}
    Let P1 be the new set of probabilities over A1
    T1 = Huffman(A1,P1)
    return T as follows: replace leaf node v in T1 by an internal node, and add two children labelled α and β below v.
```

- A 为字母表，即使用到的字符，P为每个字符出现的概率
- 从出现概率最低的两个字符开始合并，这两个字符将出现在二叉树的底端(u,r)
- 合并之后概率也将被合并
- 二叉树的末端才会有字符

9.3 Running Time

The basic steps of the Huffman algorithm are as follows:

1. Count the frequency of each symbol in the input.
2. Create a binary tree for each symbol with a weight equal to its frequency.
3. Combine the two binary trees with the lowest weights to form a new binary tree, with the sum of their weights as the weight of the new tree.
4. Repeat step 3 until all trees have been combined into a single binary tree.
5. Assign 0 or 1 to the left or right branch of each node in the binary tree, respectively, depending on whether the symbol represented by the node has been assigned a 0 or 1 bit in the code.
6. Traverse the binary tree from the root to each leaf to generate the Huffman codes for each symbol.

The time complexity of steps 1 and 2 is $O(n)$, as we need to count the frequency of each symbol and create a binary tree for each symbol. The time complexity of steps 3 and 4 is $O(nlogn)$, as we need to sort the binary trees based on their weights and merge the two trees with the lowest weights at each step, which requires $O(logn)$ time for each comparison and merge operation. The time complexity of steps 5 and 6 is also $O(nlogn)$, as we need to traverse the binary tree and assign codes to each symbol.

Therefore, the overall time complexity of the Huffman algorithm is $O(nlogn)$.

10. Segmented Least Squares

10.1 Main Idea

给出一组n个点，需要确定一条能够最好的拟合（fit）这些点的直线。有$err = \sum_{i=1}^{n}(y_i - ax_i - b)^2$ ，即点到直线的距离的平方，而这条直线具有最小的error。

使用多条直线能够做到更好的拟合，但是分成越多的段对于我们的数据分析没有意义。因为有指数中不同的划分（相当于求子集），使用暴力搜索是不现实的。

解决方法：

我们引入一个可变的变量C，作为算法的惩罚值，拟合划分的线越多，C值越大。因此我们的目标就是找到最小的：C + 各条线的$error$值。

定义$e(i, j)$ 是拟合p_i到p_j这些点时的error；$OPT(i)$为p_i到p_j的最优解（OPT(0) = 0）。此时可知：

$$OPT(n) = err(L, p_i, \ldots, p_n) + C + OPT(i-1)$$

我们无法确定i的值，但是可以选择能够给出最小值的i的值。

$$OPT(n) = \min_{1 \leq i \leq n} \{e_{i,n} + C + OPT(i-1)\}$$

10.2 Pseudo Code

```
OPTSegmentation(j)
    if (j == 0) then return
    else
        Find 1 ≤i ≤j such that M [j] = ei,j + C + M [i -1]
        OPTSegmentation(i -1)
        Output segment {pi, . . . , pj }
    end if
```

10.3 Running Time

$$T(n) = \mathrm{O}(n^2)$$

The running time of the OPTSegmentation function depends on the size of the input data and the number of segments in the optimal segmentation. Let `n` be the number of data points in the input, and `k` be the number of segments in the optimal segmentation.

The time complexity of the OPTSegmentation function is O(n^2), as it involves computing the optimal segmentation of the input data by recursively finding the best segmentation of the subproblems. The algorithm uses dynamic programming to store the cost of the optimal segmentation for all subproblems, and then uses this information to compute the optimal segmentation of the entire input data.

Here are the basic steps of the OPTSegmentation algorithm:

1. Compute the optimal segmentation of the input data by recursively finding the best segmentation of the subproblems.
2. Store the cost of the optimal segmentation for all subproblems in a matrix M.
3. Find the best segmentation of the entire input data by computing the minimum cost of all possible segmentations, using the matrix M.
4. Output the optimal segmentation of the input data.

The time complexity of step 1 is O(n^2), as we need to compute the optimal segmentation of all subproblems by recursively finding the best segmentation. The time complexity of step 2 is O(n^2), as we need to store the cost of the optimal segmentation for all subproblems in the matrix M. The time complexity of step 3 is O(n), as we need to compute the minimum cost of all possible segmentations using the matrix M. The time complexity of step 4 is O(k), as we need to output the optimal segmentation of the input data, which consists of k segments.

Therefore, the overall time complexity of the OPTSegmentation algorithm is O(n^2) for computing the optimal segmentation, and O(k) for outputting the optimal segmentation.

11. Sequence Alignment

11.1 Main Idea

- 输出一组序列和另一组序列的**最大相似结果**和最小匹配两组序列的**最小成本**。
- 两组序列的匹配可以有三种结果：匹配(matched)，空位(gap)和不匹配(mismatched)
- Matched没有匹配成本，gap有成本δ，mismatched有成本α_{ij}(假设是字母i和j)

$$OPT(i,j) = \begin{cases} j\delta & \text{, if } i = 0 \\ \min\begin{cases} \alpha_{x_i y_j} + OPT(i-1,j-1) \\ \delta + OPT(i-1,j) \\ \delta + OPT(i,j-1) \end{cases} & \text{, if } i,j \geq 1 \\ i\delta & \text{, if } j = 0 \end{cases}$$

11.2 Pseudo Code

```
SequenceAlignment(X,Y)
    Initialize M[i,0] to iδ
    Initialize M[0,j] to jδ
    for j = 1 to n do
        for i = 1 to m do
            M[i,j] = min{axiyj + M[i -1,j -1],δ + M[i -1,j],δ + M[i,j -1]}
        end for
    end for
    return M[m,n]
```

Reconstructing the optimal alignment (dynamic programming)

```
TraceAlignment(i,j)
    if i == 0 or j == 0 then return
    else
        if M[i,j] == α_{x_i y_j} + M[i -1,j -1] then
            TraceAlignment(i -1,j -1)
            Output (i,j),
        else
            if M[i,j] == δ + M[i -1,j] then TraceAlignment(i -1,j)
            else TraceAlignment(i,j -1)
            end if
        end if
    end if
```

11.3 Running Time

Two for loop: one for n and another for m, so the running time is $T(n) = O(mn)$