《数据结构与算法》课程组

重庆大学计算机学院

Data Structures & Algorithms

14

GREEDY ALGORITHM

Locally optimal choice

Outline

14.1 Activity Selection Problem

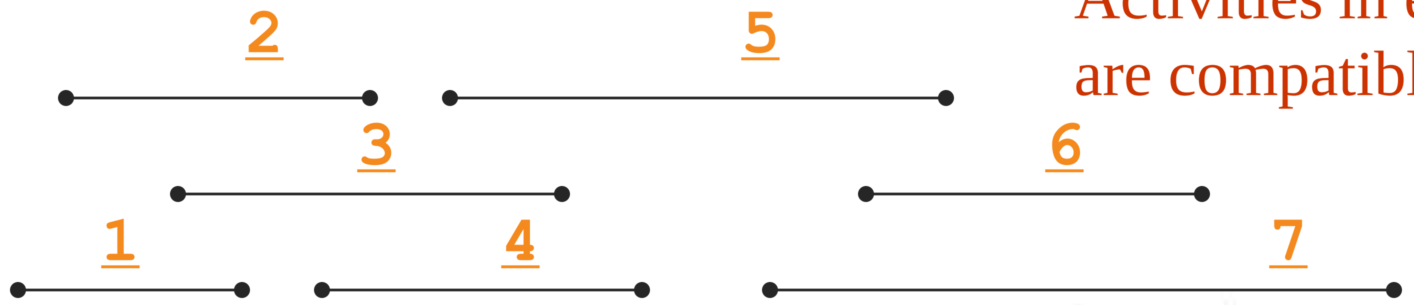
Activity-Selection Problem

- Problem: get your money’s worth out of a festival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don’t solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

Activity-selection Problem

- Input: Set S of n activities, $a_1, a_2, \ldots, a_n$.
  - s_i = start time of activity i.
  - f_i = finish time of activity i.
- Output: Subset A of maximum number of compatible activities.
  - Two activities are compatible, if their intervals don't overlap.

Example:



Activities in each line are compatible.

Optimal Substructure

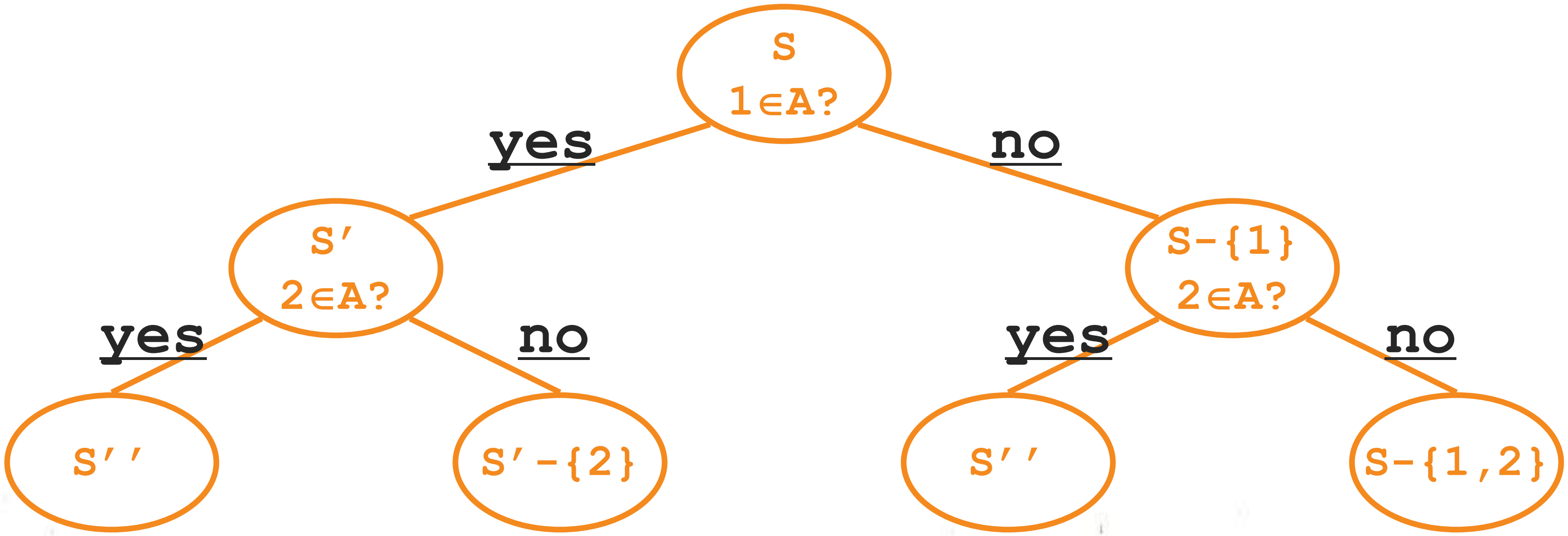
- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - This generates two subproblems.
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - The solutions to the two subproblems must be optimal.
    - Prove using the cut-and-paste approach.

Optimal Substructure

- Assume activities are sorted by finishing times.
  - $f_1 \leq f_2 \leq \ldots \leq f_n$.
- Suppose an optimal solution includes activity a_k.
  - This generates two subproblems.
  - Selecting from $a_1, \ldots, a_{k-1}$, activities compatible with one another, and that finish before a_k starts (compatible with a_k).
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes.
  - The solutions to the two subproblems must be optimal.
    - Prove using the cut-and-paste approach.

Activity Selection: Repeated Subproblems

- Consider a recursive algorithm that tries all possible compatible subsets to find a maximal set, and notice repeated subproblems:



Recursive Solution

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts.
- Subproblems: Selecting maximum number of mutually compatible activities from S_{ij}.
- Let $c[i, j]$ = size of maximum-size subset of mutually compatible activities in S_{ij}.

Recursive Solution:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \phi \\ \max\limits_{i<k<j}\{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \phi \end{cases}$$

Greedy Choice Property

- Dynamic programming? Memoize? Yes, but…
- Activity selection problem also exhibits the *greedy choice* property:
  - Locally optimal choice $\Rightarrow$ globally optimal solution
  - Theorem: if S is an activity selection problem sorted by finish time, then $\exists$ optimal solution $A \subseteq S$ such that $\{1\} \in A$
    - Sketch of proof: if $\exists$ optimal solution B that does not contain {1}, can always replace first activity in B with {1} (*Why?*). Same number of activities, thus optimal.

Greedy-choice Property

- The problem also exhibits the greedy-choice property.
  - There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}.
  - Can be proved easily.
- Hence, there is an optimal solution to S that includes a_1.
- Therefore, make this greedy choice without solving subproblems first and evaluating them.
- Solve the subproblem that ensues as a result of making this greedy choice.
- Combine the greedy choice and the solution to the subproblem.

Recursive Algorithm

```
Recursive-Activity-Selector ($s, f, i, j$)
1.  $m \leftarrow i+1$
2.  while $m < j$ and $s_m < f_i$
3.      do $m \leftarrow m+1$
4.  if  $m < j$
5.      then return $\{a_m\} \cup$
                Recursive-Activity-Selector($s, f, m, j$)
6.      else return $\phi$
```

Initial Call: Recursive-Activity-Selector (s, f, 0, n+1)

Complexity: $\Theta(n)$

Straightforward to convert the algorithm to an iterative one.

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.
- Make the greedy choice and **solve top-down**.
- May have to preprocess input to put it into greedy order.
  - Example: Sorting activities by finish time.

Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Intuition is even more simple:
  - Always pick the shortest ride available at the time

```
GREEDY-ACTIVITY-SELECTOR($s$, $f$)
n ← length[s]
A ← {a_1}
i ← 1
for m ← 2 to n
        do if $s_m \geq f_i$
                then $A \leftarrow A \cup \{a_m\}$
                        $i \leftarrow m$
return A
```

14.2 Elements of Greedy Algorithm

Overview

- Like dynamic programming, used to solve optimization problems.
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code
- Problems exhibit optimal substructure (like DP).
- Problems also exhibit the **greedy-choice** property.
  - When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

Elements of Greedy Algorithms

- Greedy-choice Property.
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- Optimal Substructure.

Greedy Strategy

- The choice that seems best at the moment is the one we go with.
  - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it’s always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.

思考题

礼品分组

N个礼品，每个礼品的价格不一样。现要把所有礼品分组，每组的礼品数量不超过2个，且礼品总价格不超过C (C>0)，求分组的数目最少的分法。

思考题

堆积木

老师给每个小朋友分了些积木块，但每个小朋友手上的积木都不足以堆成想要的形状。现在你手上有一些积木，你可以全部交给某个小朋友让他有足够的积木堆成形状，堆完后再收回所有的积木。你最多可以让多少小朋友堆成积木。

思考题

PK赛

土木系男生寝室A和煤矿系男生寝室B的人数都是N，为了争夺和艺术系女生寝室的“联谊”权，决定举行一场扳手腕PK赛。比赛要进行N轮，每轮由双方寝室各派出一位男生参加，但每人只能比赛一次。假设寝室B的室长知道双方学生的实力，他如何安排寝室B学生的比赛顺序才能取得最多的胜利。

经典面试题

高速公路上设置有n座加油站，位置分别为 $x_1 < x_2 \ldots < x_n$，
加油站x_j的每单位油的价格为B_j $(1 \le j \le n)$。
车的油箱容量为P，每单位油可以跑1的距离。
最开始车在加油站x_1 且油箱为空，为了到达加油站x_n，
需要在途中的加油站加油，求支付出的油价总额最少的加油方法
（$\forall j \in \{1,..,n-1\}$: $x_{j+1} - x_j < P$）

经典面试题

高速公路上设置有n座储油罐，位置分别为 $0 < x_1 < x_2 \ldots < x_n$，在 x_j 的储油罐装有 B_j 的油$(1 \le j \le n)$。

车的油箱足够大，可以装完所有油罐的油。

开始时车在起点$x_o = 0$，装有P的油，每单位油可以跑1的距离，途中经过的油罐里的油可以任意取，也可以不取。

$$\forall j \in \{1, \ldots, n\}: x_j \le \mathrm{P} + \sum_{t=0}^{j-1} B_t$$

目的地离起点有 L 的距离且 $L \le P + \sum_{j=1}^{n} B_j$ 。

求驾车到达目的地，取油次数最少的方法

14.3 Huffman Codes

Data Compression

Q. Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?

Q. Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

Q. How do we know when the next symbol begins?

Ex. $c(a) = 01$ What is 0101?

$c(b) = 010$

$c(e) = 1$

Data Compression

Q. Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?
A. We can encode 32 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Q. Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?
A. Encode these characters with fewer bits, and the others with more bits.

Q. How do we know when the next symbol begins?
A. Use a separation symbol (like the pause in Morse), or make sure that there is no ambiguity by ensuring that no code is a prefix of another one.

Ex. c(a) = 01 What is 0101?
c(b) = 010
c(e) = 1

Prefix Codes

Definition. A prefix code for a set S is a function c that maps each x∈S to 1s and 0s in such a way that for x,y∈S, x≠y, c(x) is not a prefix of c(y).

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000

Q. What is the meaning of 1001000001 ?
A. “leuk”

Suppose frequencies are known in a text of 1G:
fa=0.4, fe=0.2, fk=0.2, fl=0.1, fu=0.1
Q. What is the size of the encoded text?
A. 2*fa + 2*fe + 3*fk + 2*fl + 4*fu = 2.4G

Optimal Prefix Codes

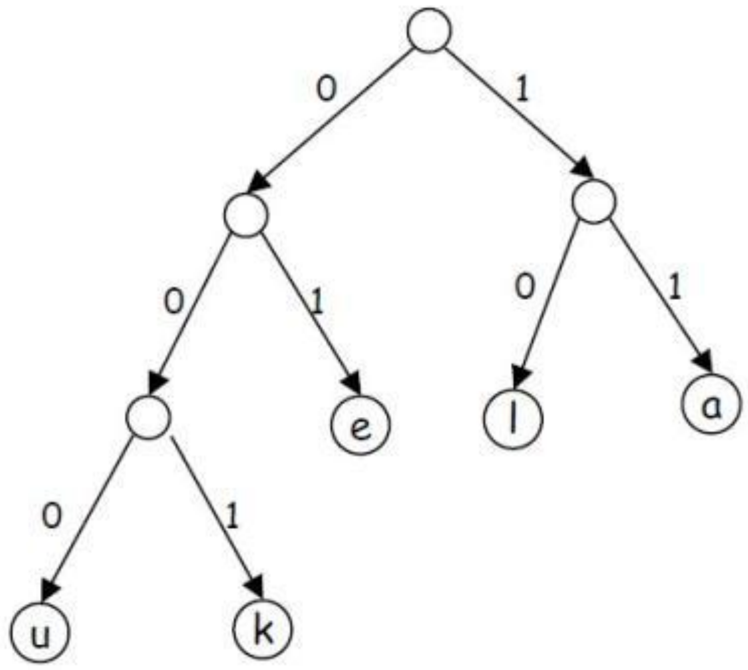
Definition. The average bits per letter of a prefix code c is the sum over all symbols of its frequency times the number of bits of its encoding:

$$ABL(c) = \sum_{x \in S} f_x \cdot |c(x)|$$

We would like to find a prefix code that is has the lowest possible average bits per letter.
Suppose we model a code in a binary tree…

Representing Prefix Codes using Binary Trees

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000

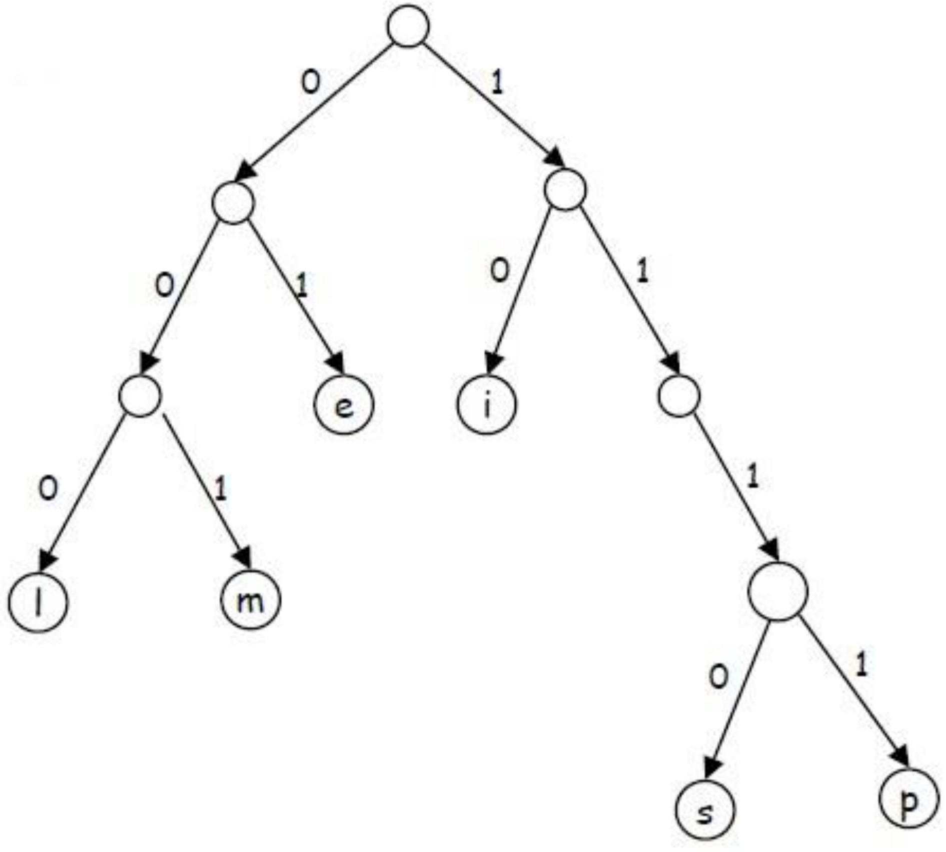


Q. How does the tree of a prefix code look?
A. Only the leaves have a label.
Pf. An encoding of x is a prefix of an encoding of y if and only if the path of x is a prefix of the path of y.

Representing Prefix Codes using Binary Trees

Q. What is the meaning of
111010001111101000 ?
A. “simpel”

$$ABL(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x)$$



Q. How can this prefix code be made more efficient?

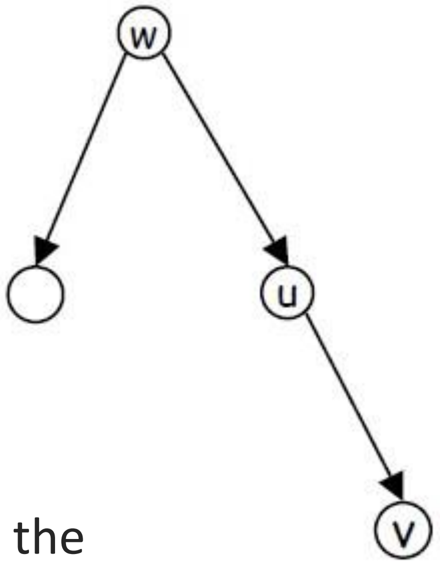
A. Change encoding of p and s to a shorter one.
This tree is now full.

Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

Claim. The binary tree corresponding to the optimal prefix code is full.

Proof. (by contradiction)
Suppose T is binary tree of optimal prefix code and is not full.
This means there is a node u with only one child v.

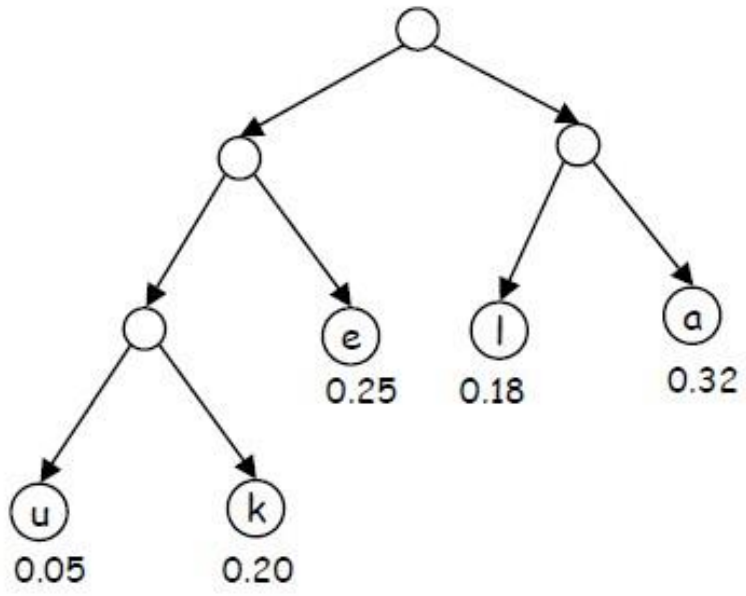


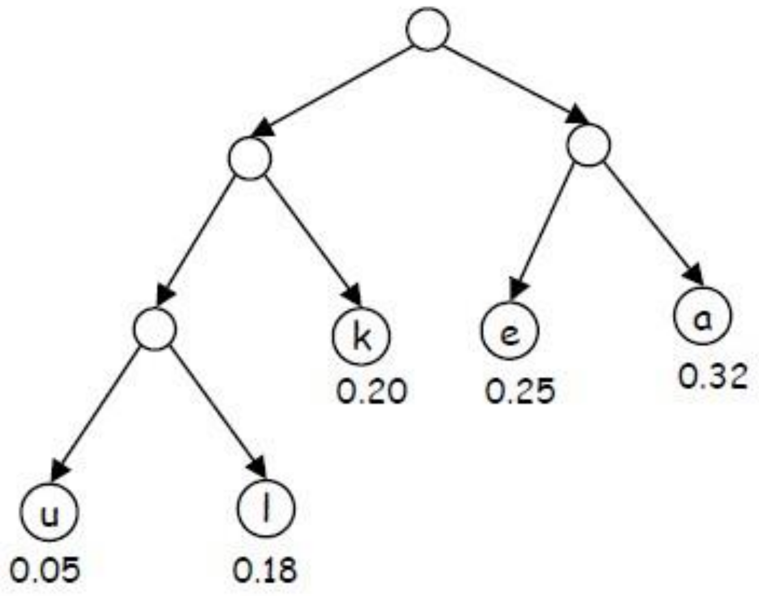
- Case 1: u is the root; delete u and use v as the root
- Case 2: u is not the root
  - let w be the parent of u
  - delete u and make v be a child of w in place of u
- In both cases the number of bits needed to encode any leaf in the subtree of v is decreased. The rest of the tree is not affected.
- Clearly this new tree T' has a smaller ABL than T. Contradiction.

Optimal Prefix Codes: False Start

Q. Where in the tree of an optimal prefix code should letters be placed with a high frequency?
A. Near the top.

Greedy template 1: Create tree top-down, split S into two sets S1 and S2 with (almost) equal frequencies. Recursively build tree for S1 and S2. [Shannon-Fano, 1949] f_a=0.32, f_e=0.25, f_k=0.20, f_l=0.18, f_u=0.05





Optimal Prefix Codes: Huffman Encoding

Observation 1: Lowest frequency items should be at the lowest level in tree of optimal prefix code.

Observation 2: For n > 1, the lowest level always contains at least two leaves.

Observation 3: The order in which items appear in a level does not matter.

Claim. There is an optimal prefix code with tree T* where the two lowest-frequency letters are assigned to leaves that are siblings in T*.

Greedy template 2: [Huffman, 1952] Create tree bottom-up.
Make two leaves for two lowest-frequency letters y and z.
Recursively build tree for the rest using a meta-letter for yz.

Building Human Trees

- Create a collection of n initial Huffman trees, each of which is a single leaf node containing one of the letters. Put the n partial trees onto a list in ascending order by weight (frequency).
- Next, remove the first two trees (the ones with lowest weight) from the list. Join these two trees together to create a new tree whose root has the two trees as children, and whose weight is the sum of the weights of the two trees. Put this new tree back on the list in the correct place necessary to preserve the order of the list.
- This process is repeated until all of the partial Huffman trees have been combined into one.

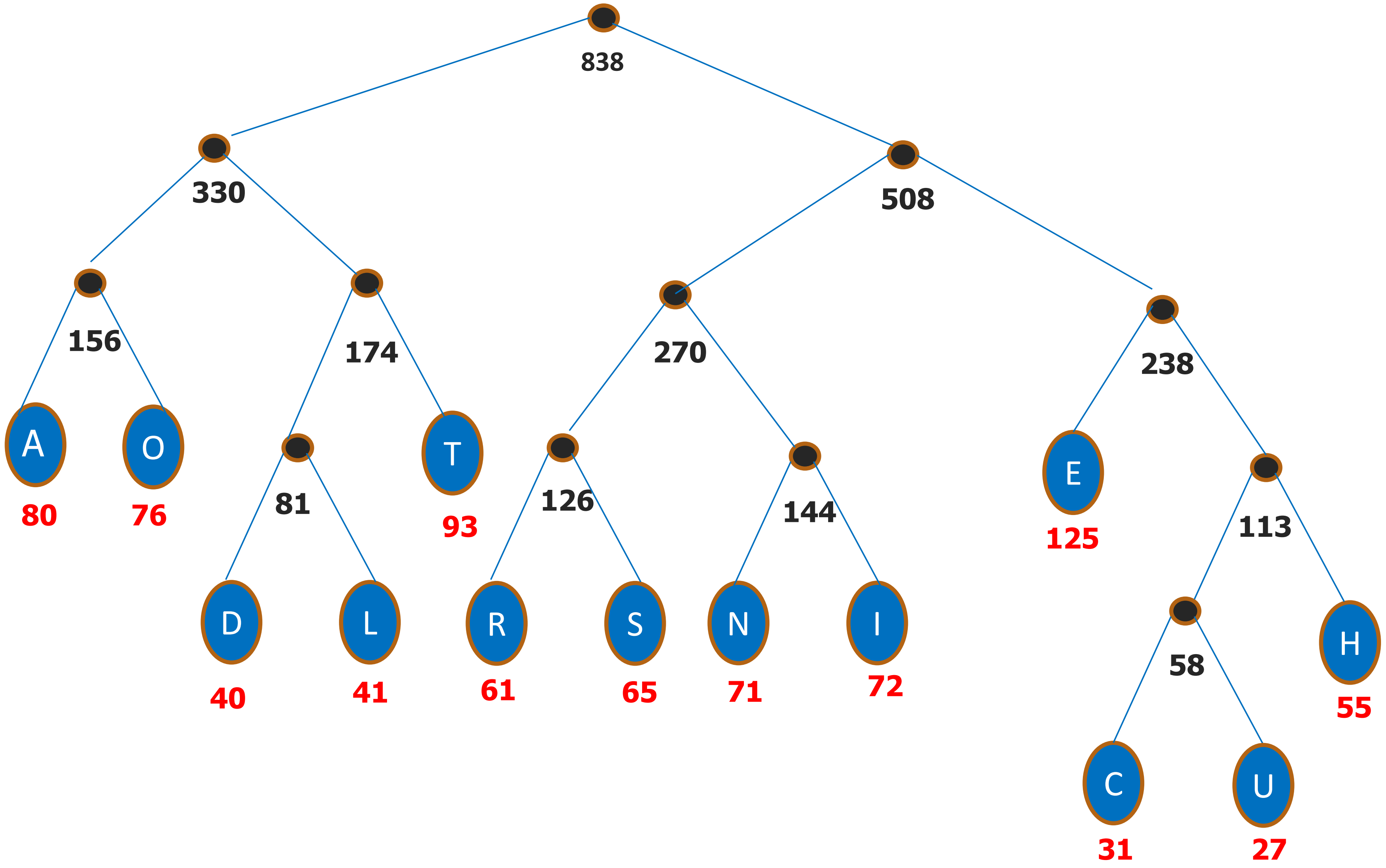
Example

- Character count in text:

Char	E	T	A	O	I	N	S	R	H	L	D	C	U
Freq	125	93	80	76	72	71	65	61	55	41	40	31	27

- At first, there are 13 partial trees.

Huffman Tree Construction



Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.
Proof. (by induction)
Base: For n=2 there is no shorter code than root and two leaves.
Hypothesis: Suppose Huffman tree T' for S' of size n-1 with ω instead of y and z is optimal. (IH)
Step: (by contradiction)

- Idea of proof:

– Suppose other tree Z of size n is better.
– Delete lowest frequency items y and z from Z creating Z'
– Z' cannot be better than T' by IH.

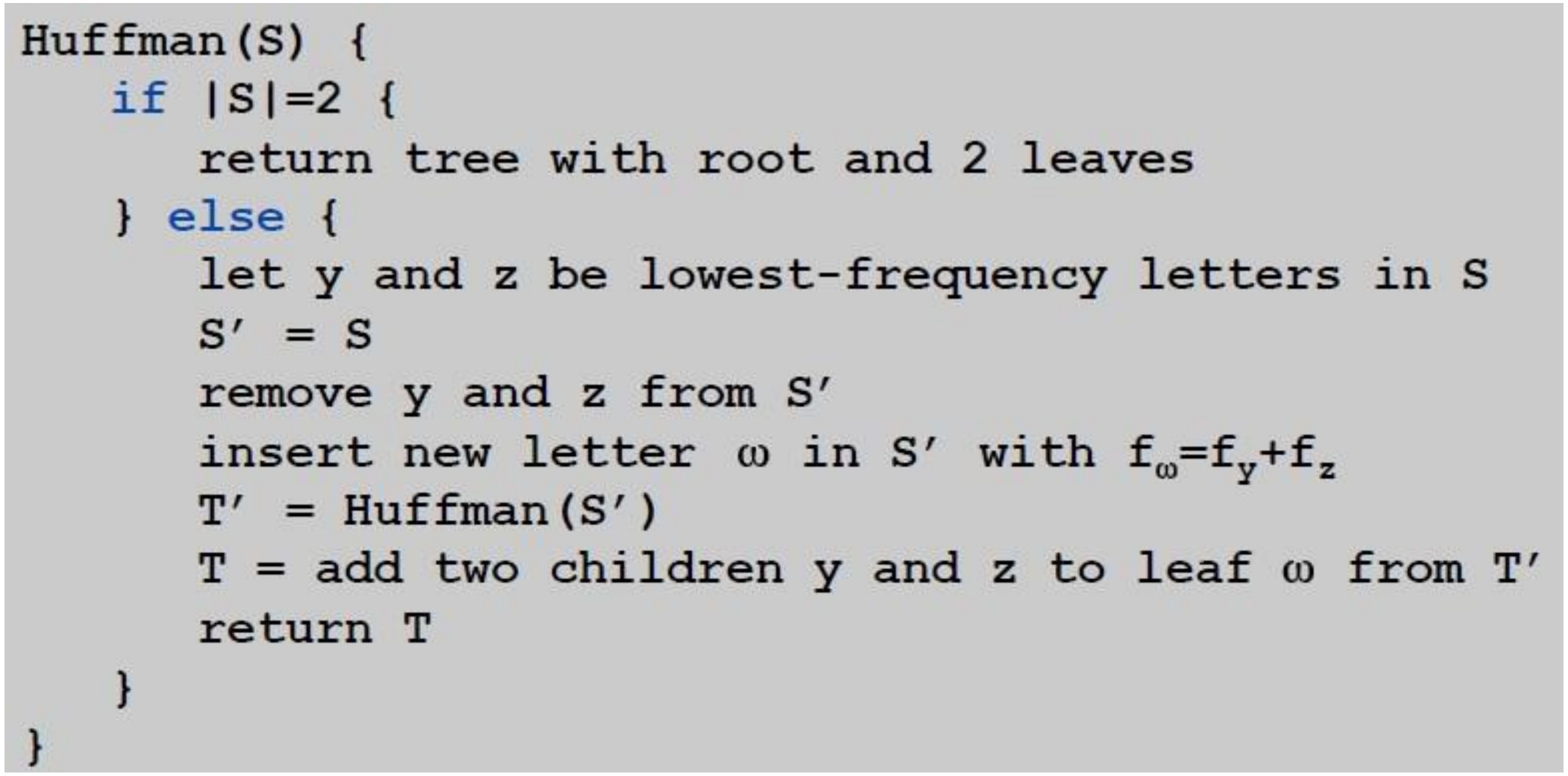
Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.
Proof. by induction, based on optimality of T' (y and z removed, ω added)

Claim. ABL(T')=ABL(T)-f_ω
Pf.

$$\begin{aligned}
\mathrm{ABL}(T) &= \sum_{x\in S} f_x \cdot \mathrm{depth}_T(x) \\
&= f_y \cdot \mathrm{depth}_T(y) + f_z \cdot \mathrm{depth}_T(z) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= \left(f_y + f_z\right) \cdot \left(1 + \mathrm{depth}_T(\omega)\right) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= f_\omega \cdot \left(1 + \mathrm{depth}_T(\omega)\right) + \sum_{x\in S, x\neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= f_\omega + \sum_{x\in S'} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_\omega + \mathrm{ABL}(T')
\end{aligned}$$

Optimal Prefix Codes: Huffman Encoding



```
Huffman(S) {
   if |S|=2 {
      return tree with root and 2 leaves
   } else {
      let y and z be lowest-frequency letters in S
      S′ = S
      remove y and z from S′
      insert new letter ω in S′ with $f_\omega = f_y + f_z$
      T′ = Huffman(S′)
      T = add two children y and z to leaf ω from T′
      return T
   }
}
```

Q. What is the time complexity?

A. $T(n) = T(n-1) + O(n)$ so $O(n^2)$

Q. How to implement finding lowest-frequency letters efficiently?

A. Use priority queue (min heap) for S: $T(n) = T(n-1) + O(\log(n))$ so $O(n \log(n))$

Implementation: Huffman Tree Nodes(1)

```
template <class Elem>
class HuffNode {    //Node abstract base class
public:
   virtual int weight() = 0;
   virtual bool isLeaf() = 0;
   virtual HuffNode* left() const = 0;
   virtual void setLeft(HuffNode*) = 0;
   virtual HuffNode* right() const = 0;
   virtual void setRight(HuffNode*) = 0;
};
```

Implementation: Huffman Tree Nodes(2)

```
template <class Elem>     //leaf node subclass
class LeafNode: public HuffNode<Elem> {
private:
   Freqpair<Elem>* it;       //Frequency pair
public:
   LeafNode(const Elem& val, int freq)     //constructor
      { it = new Freqpair<Elem>(val,freq);  }
   int weight()   { return it->weight(); }   //Return frequency
   Freqpair<Elem>* val()   {  return it;  }
   bool isLeaf()   { return true; }
   virtual HuffNode* left() const  { return NULL; }
   virtual void setLeft (HuffNode*)  {  }
   virtual HuffNode* right() const  { return NULL; }
   virtual void setRight(HuffNode*)  {  }
};
```

Implementation: Huffman Tree Nodes(3)

```
template <class Elem>     //Internal node subclass
class IntlNode: public HuffNode<Elem> {
private:
   HuffNode<Elem>*  lc;          //left child
   HuffNode<Elem>*  rc;          //right child
   int wgt;                      //Subtree weight
public:
   IntlNode(HuffNode<Elem> *  l; HuffNode<Elem> * r)
     { wgt = l->weight() + r->weight();   lc = l;  rc = r; }
   int weight()    { return  wgt; }     //Return frequency
   bool isLeaf()   { return  false; }
   HuffNode<Elem>* left() const  { return  lc; }
   void setLeft(HuffNode<Elem>* b)
     { lc = (HuffNode*)b;  }
   HuffNode<Elem>* right() const  { return  rc; }
   void setRight(HuffNode<Elem>* b)
     { rc = (HuffNode*)b;  }
};
```

Class Declaration: Frequency Pair Object

```
template <class Elem>
class FreqPair  {     //An element / frequency pair
private:
   Elem it;               //An element of some sort
   int freq;
public:
   FreqPair(const Elem& e, int f)    //Constructor
     {  it = e;     freq = f;   }
   ~FreqPair() {  }                          //Destructor
   int weight()  {  return  freq; }  //Return the weight
   Elem& val()  {  return  it;   }    //Return the element
};
```

Class Declaration: Huffman Tree

```
template <class Elem>
class  HuffTree  {
private:
   HuffNode<Elem>*  theRoot;
public:
   HuffTree(Elem& val,  int freq)
      { theRoot = new LeafNode<Elem>(val,freq);  }
   HuffTree(HuffTree<Elem>*  l, HuffTree<Elem>*  r)
      { theRoot = new IntlNode<Elem>(l->root(), r->root()); }
   ~ HuffTree() {  }
   HuffNode<Elem>* root()  {  return  theRoot;  }
   int weight()   {  return  theRoot->weight();   }
};
```

Class Declaration: Huffman Tree(2)

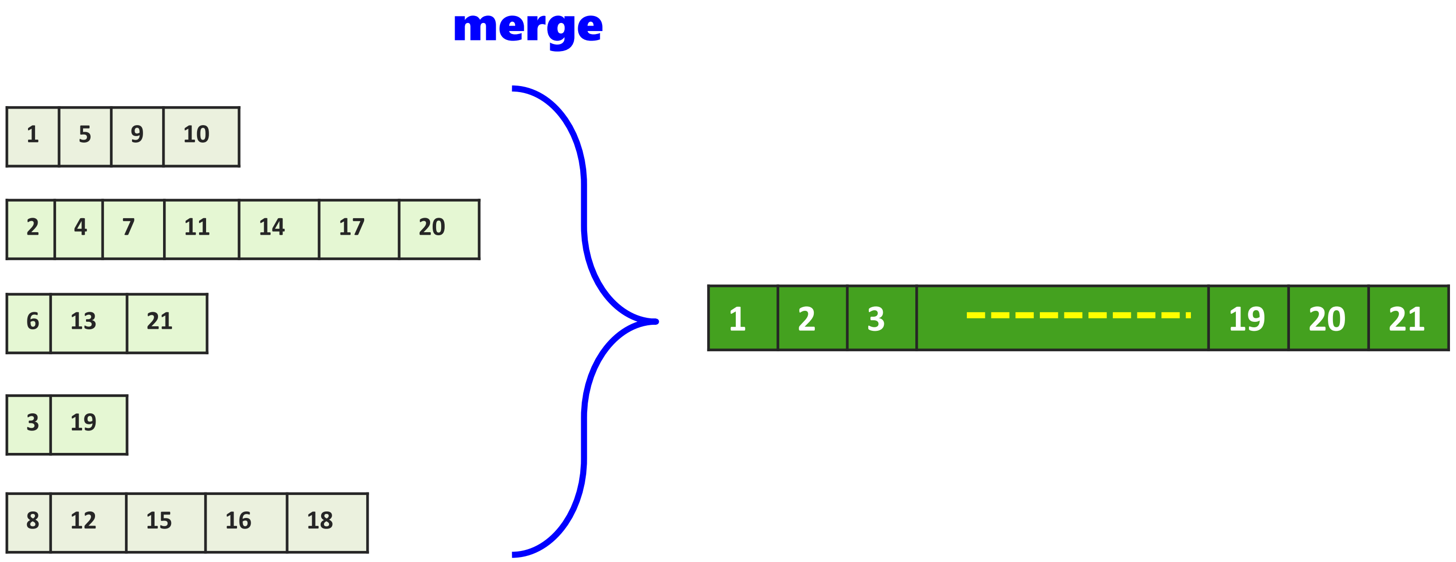
```
template <class Elem>  class HHCompare  {
public:
  static bool lt (HuffTree<Elem>* x, HuffTree<Elem>*  y)
    { return x->weight() < y->weight();  }
  static bool eq(HuffTree<Elem>* x, HuffTree<Elem>*  y)
    { return x->weight() = = y->weight();  }
  static bool gt(HuffTree<Elem>* x, HuffTree<Elem>*  y)
    { return x->weight() > y->weight();  }
};
```

Huffman Tree Construction

```
template <class Elem>  HuffTree<Elem>*
buildHuff(SLList<HuffTree<Elem>*,HHCompare<Elem> >* f1) {
   HuffTree<Elem>* temp1, *temp2, *temp3;
   for (f1->setStart(); f1->leftLength()+f1->rightLength()>1;
        f1->setStart())  {          //While at least two items left
      f1->remove(temp1);        //Pull first two trees off the list
      f1->remove(temp2);
      temp3 = new HuffTree<Elem>(temp1, temp2);
      f1->insert(temp3);            //Put the new tree back on list
      delete temp1;                 //Must delete the remnants
      delete temp2;                 //   of the trees we created
   }
   return temp3;
}
```

思考题：合并n个升序或降序序列

把n个升序（降序）序列：$A_1, A_2, \ldots, A_n$，合并成一个升序（降序）序列。

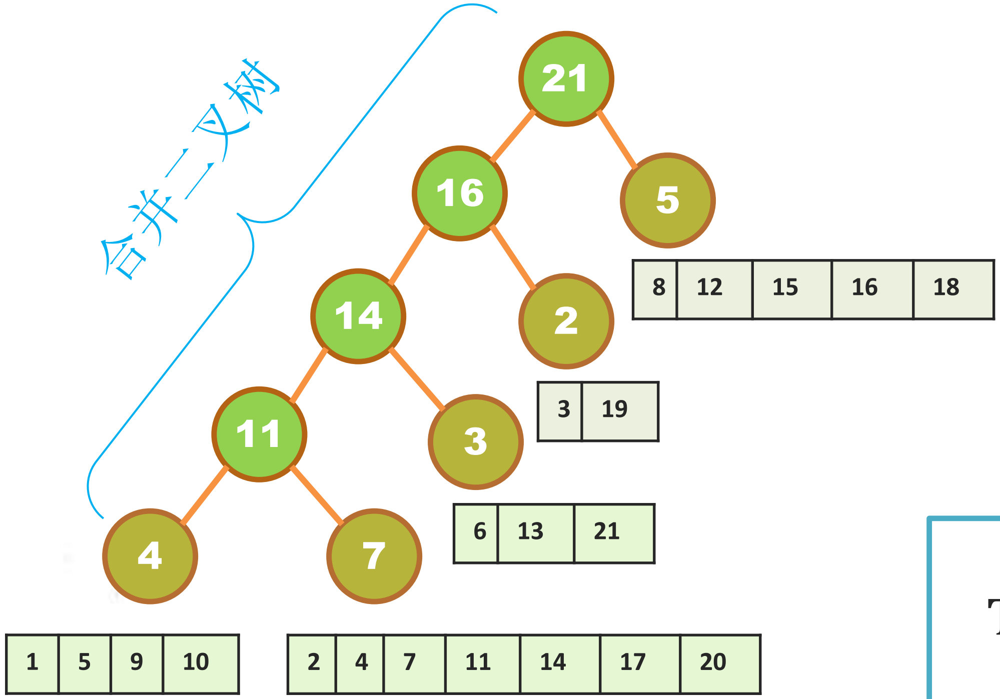


合并n个升序或降序序列

*方法1：顺序合并

- 可用N个叶节点的(Full)二叉树表达合并过程

- 比较总次数：62



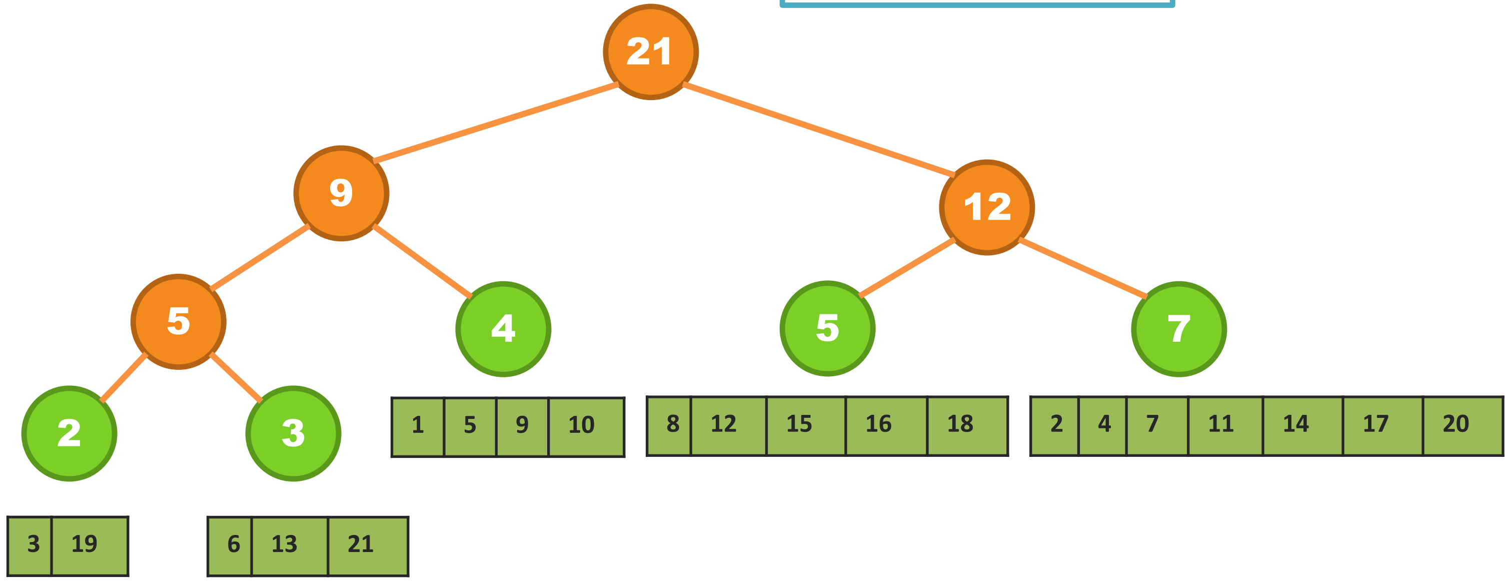
- 中间节点的权重表示两个序列合并后的长度及合并时间（比较次数）
- 合并总时间等于中间节点的权重和

$$T\left(\sum_{1\le i\le n}|A_i|\right)=\sum_{i=1}^{n}Depth(A_i)*|A_i|$$

Huffman algorithm: application

*方法2：构建哈夫曼树 (why?)

比较次数：47



$$\mathrm{T}\left(\sum_{1\le i\le n}|A_i|\right)=\sum_{i=1}^{n} Depth(A_i)*|A_i|+N*\log(N)$$

数据结构与算法课程组

重庆大学计算机学院

End of Section.