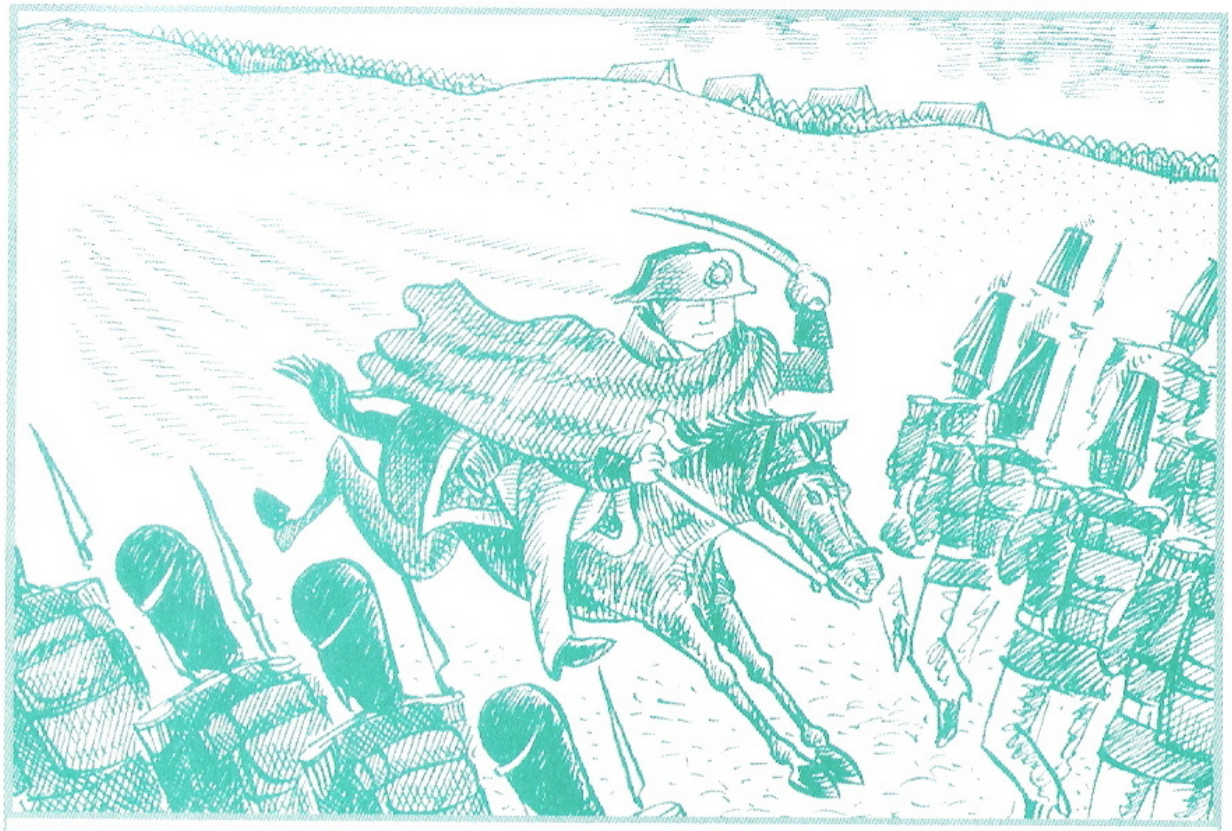
Algorithms

MIS, KUAS, 2007

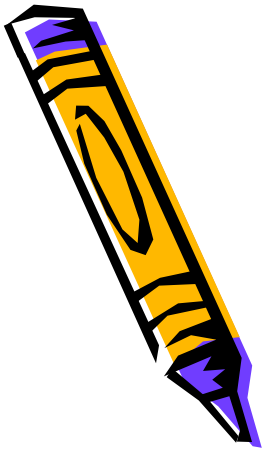
Jen-Wen Ding



chapter 2

Divide-and-Conquer

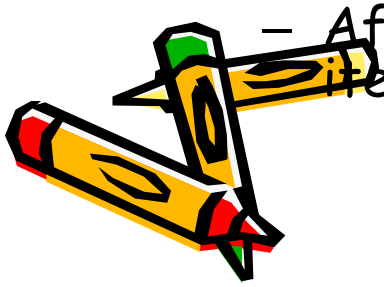
2 What is Divide-and-Conquer?



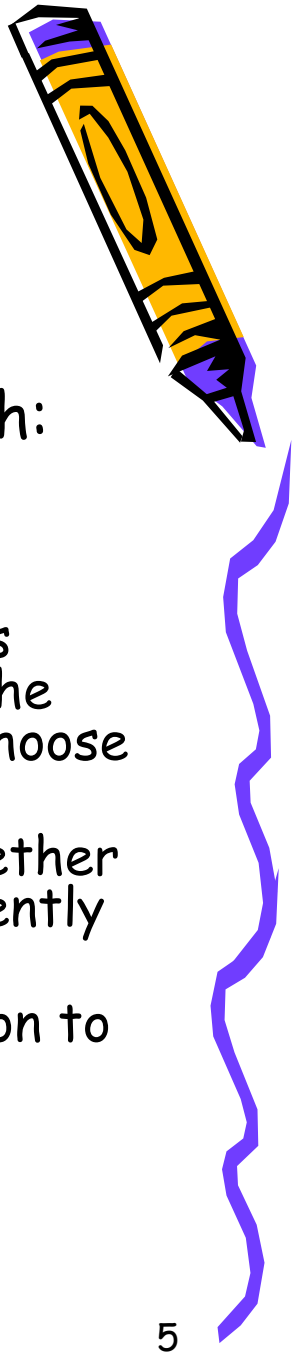
- Divide-and-Conquer divides an instance of a problem into two or more smaller instances.
  - The smaller instances are usually instances of the original problem.
- If solutions to the smaller instances can be obtained readily, the solution to the original instance can be obtained by combining these solutions.
- If the smaller instances are still too large to be solved readily, they can be divided into still smaller instances.
  - This process of dividing the instances continues until they are so small that a solution is readily obtainable.

Overview of Divide-and-Conquer

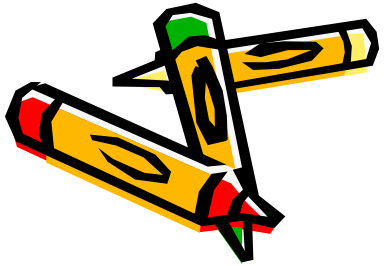
- The divide-and-conquer approach is a ***top-down*** approach.
  - That is, the solution to a *top-level* instance of a problem is obtained by going *down* and obtaining solutions to smaller instances.
- The reader may recognize this as the method used by recursive routines.
- Recall that when writing recursion, one thinks at the problem-solving level and lets the system handle the details of obtaining the solution (by means of stack manipulations).
- When developing a divide-and-conquer algorithm, we usually think at this level and write it as a recursive routine.
  - After this, we can sometimes create a more efficient iterative version of the algorithm.



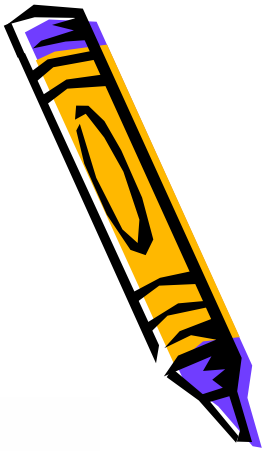
2.1 Binary Search



- The divide-and-conquer steps of Binary Search:
- If *x* equals the middle item, quit.
- Otherwise:
  - *(1) Divide* the array into two subarrays about half as large. If *x* is smaller than the middle item, choose the left subarray. If *x* is larger than the middle item, choose the right subarray.
  - *(2) Conquer* (solve) the subarray by determining whether *x* is in that subarray. Unless the subarray is sufficiently small, use recursion to do this.
  - *(3) Obtain* the solution to the array from the solution to the subarray.



Example of Binary Search



Example 2.1

Suppose x = 18 and we have the following array:

10 12 13 14 18 20 25 27 30 35 40 45 47.

↑

Middle item

1. Divide the array: Because x <25, we need to search

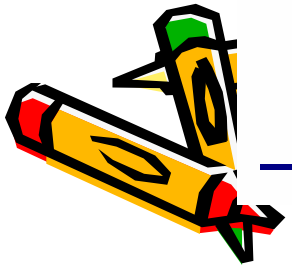
   10 12 13 14 18 20.

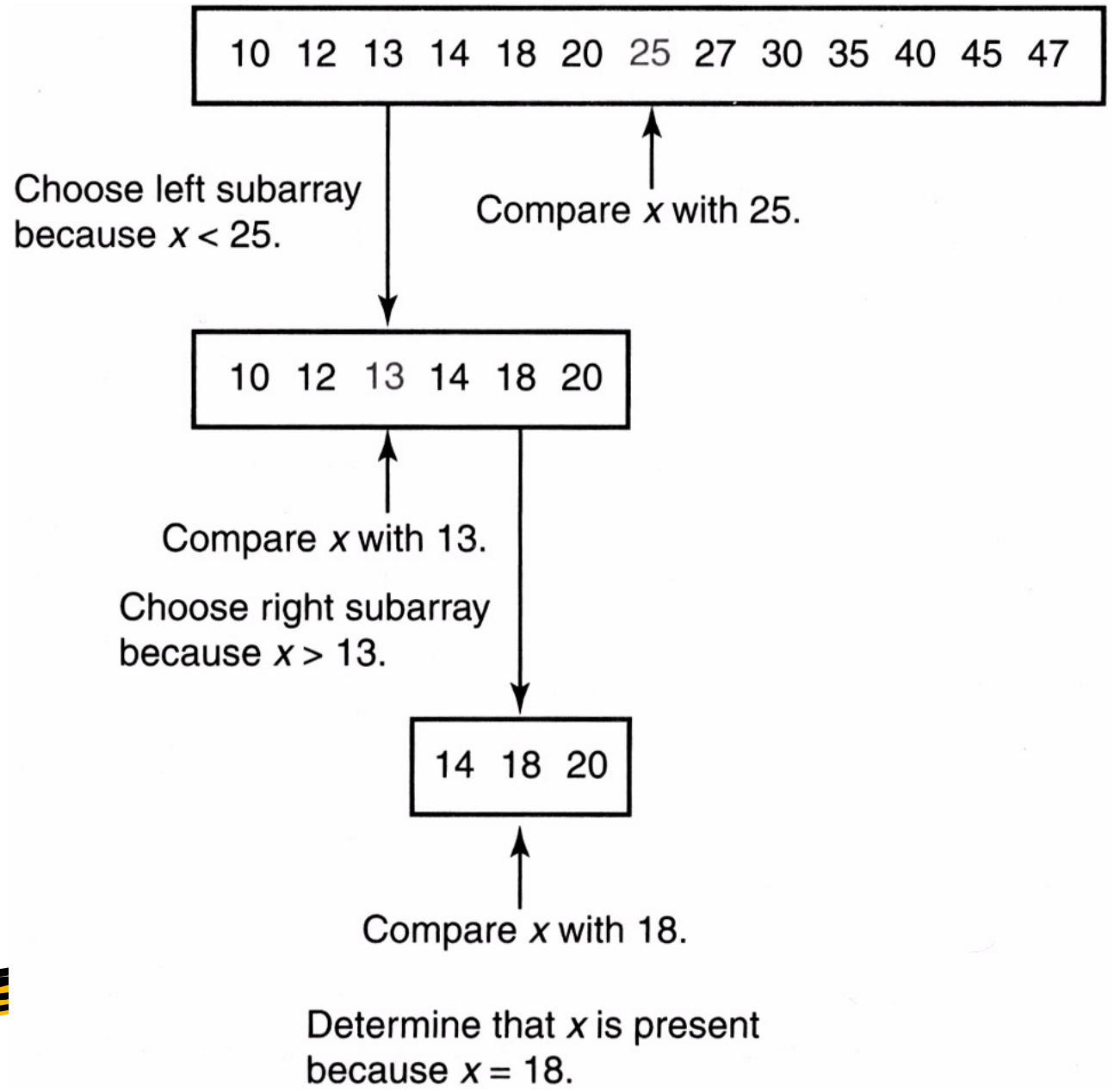
2. Conquer the subarray by determining whether x is in the subarray. This is accomplished by recursively dividing the subarray. The solution is:

   Yes, x is in the subarray.

3. Obtain the solution to the array from the solution to the subarray:

   Yes, x is in the array.




10 12 13 14 18 20 25 27 30 35 40 45 47
Choose left subarray because x < 25.
Compare x with 25.
10 12 13 14 18 20
Compare x with 13.
Choose right subarray because x > 13.
14 18 20
Compare x with 18.
Determine that x is present because x = 18.

Algorithm 2.1: Binary Search (Recursive)

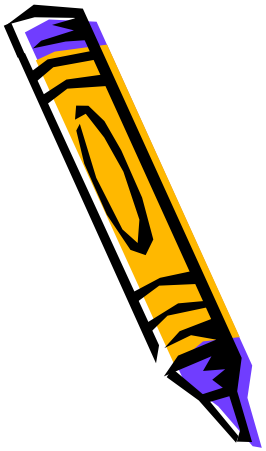
Problem: Determine whether x is in the sorted array S of size n.

Inputs: positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.

Outputs: *location*, the location of x in S (0 if x is not in S).

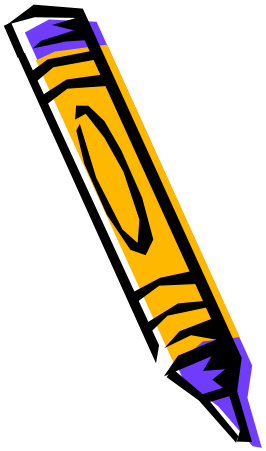
```
index location (index low, index high)
   {
  index mid;
  if (low > high)
        return 0;
  else {
        mid = [(low + high)/2];
        if (x == S[mid])
              return mid
        else if (x < S[mid])
              return location(low, mid - 1);
        else
              return location(mid + 1, high);
     }
   }
```

Comment for Algorithm 2.1

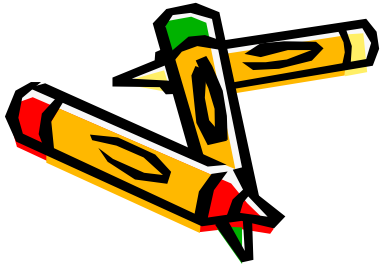


- Notice that *n*, *S*, and *x* are not parameters to function *location*.
- Because they remain unchanged in each recursive call, there is no need to make them parameters.
- There are two reasons for doing this.
  - First, it makes the expression of recursive routines less cluttered.
  - Second, in an actual implementation of a recursive routine, a new copy of any variable passed to the routine is made in each recursive call. If a variable's value does not change, the copy is unnecessary. This waste could be costly if the variable is an array.

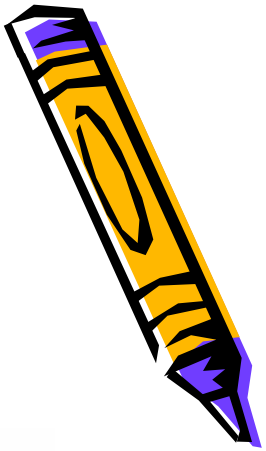
Analysis of Algorithm 2.1 Worst-Case Time Complexity (Binary Search, Recursive)



- Basic operation
  - the comparison of *x* with *S[mid]*
- Input size: *n*
  - the number of items in the array
- There are two comparisons of *x* with *S[mid]* in any call to function *location* in which *x* does not equal *S[mid]*.
  - we can assume that there is only one comparison, because this would be the case in an efficient assembler language implementation



Analysis of Algorithm 2.1 Worst-Case Time Complexity (Binary Search, Recursive)



$$W(n) = \underbrace{W\left(\frac{n}{2}\right)}_{\text{Comparisons in recursive call}} + \underbrace{1}_{\text{Comparison at top level}}$$

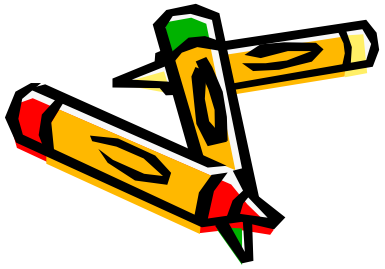
$$W(n) = W\left(\frac{n}{2}\right) + 1 \qquad \text{for } n > 1,\ n \text{ a power of 2}$$
$$W(1) = 1$$

This recurrence is solved in Example B.1 in Appendix B. The solution is

$$W(n) = \lg n + 1.$$

If n is not restricted to being a power of 2, then

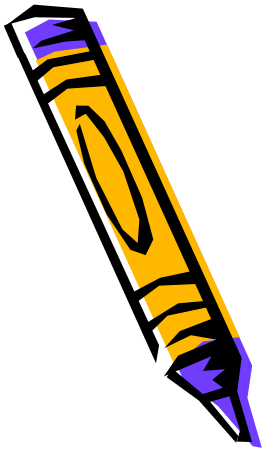
$$W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$$



2.2 Mergesort

- The divide-and-conquer steps of Mergesort
  - *(1) Divide* the array into two subarrays each with $n/2$ items.
  - *(2) Conquer* (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.
  - *(3) Combine* the solutions to the subarrays by merging them into a single sorted array.

Example of Mergesort



Example 2.2

Suppose the array contains these numbers in sequence:

27 10 12 25 13 15 22.

1. Divide the array:

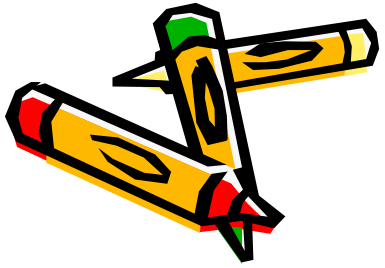
   27 10 12 20 and 25 13 15 22.

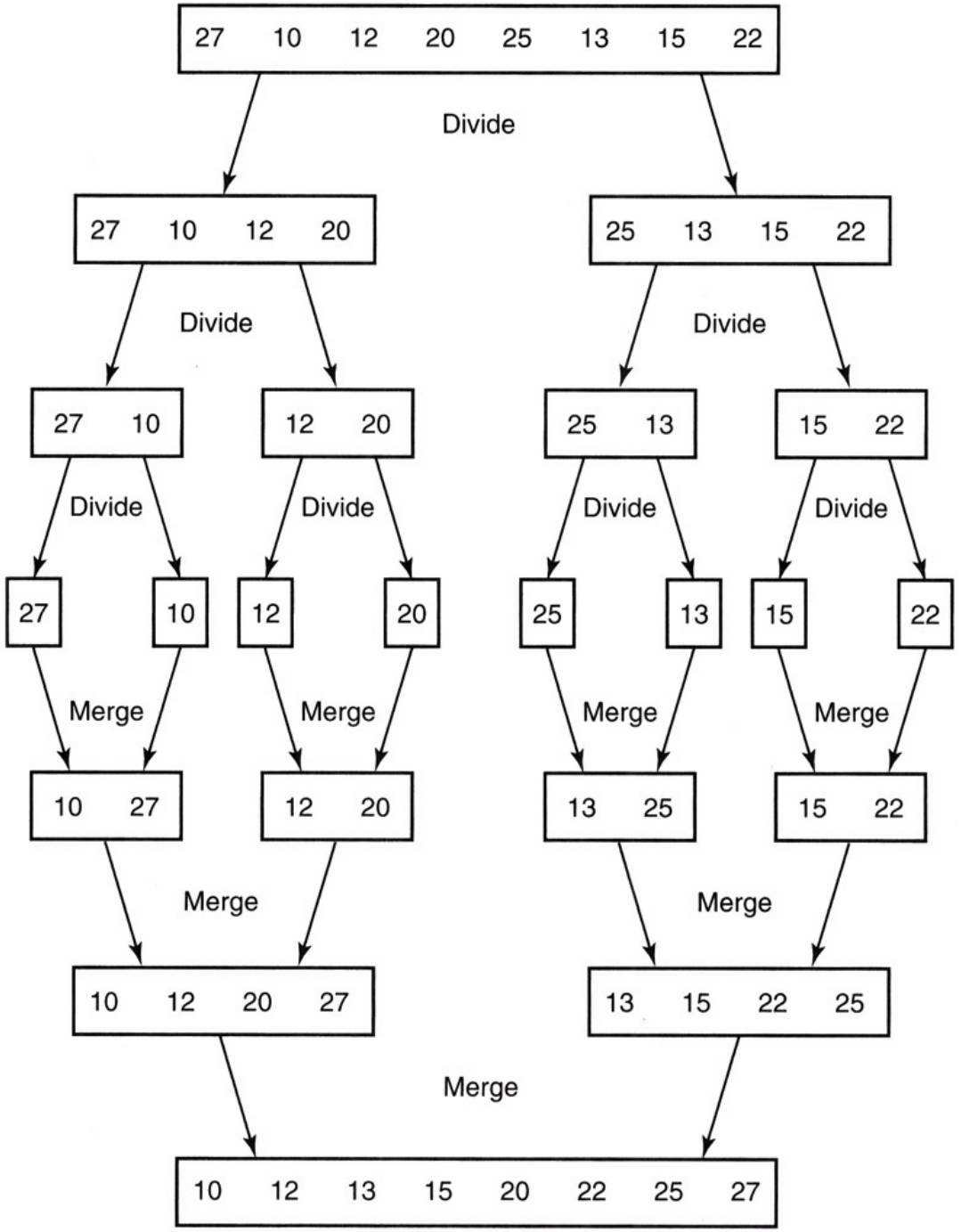
2. Sort each subarray:

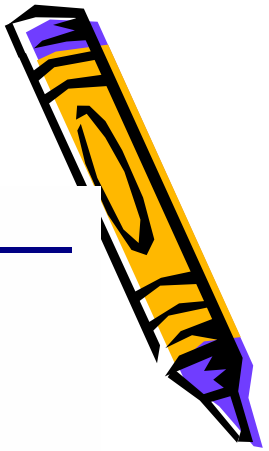
   10 12 20 27 and 13 15 22 25.

3. Merge the subarrays:

   10 12 13 15 20 22 25 27.




27 10 12 20 25 13 15 22
Divide
27 10 12 20
25 13 15 22
Divide
Divide
27 10
12 20
25 13
15 22
Divide
Divide
Divide
Divide
27
10
12
20
25
13
15
22
Merge
Merge
Merge
Merge
10 27
12 20
13 25
15 22
Merge
Merge
10 12 20 27
13 15 22 25
Merge
10 12 13 15 20 22 25 27



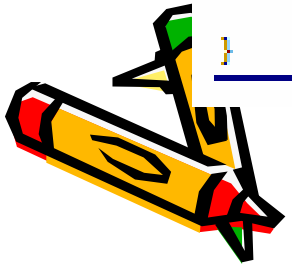
Algorithm 2.2: Mergesort

Problem: Sort *n* keys in nondecreasing sequence.

Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*.

Outputs: the array *S* containing the keys in nondecreasing order.

```
void mergesort (int n, keytype S[])
{
     if (n>1) {
          const int h=[n/2], m = n - h;
          keytype U[1 ..h], V[1 ..m];
          copy S[1] through S[h] to U[1] through U[h];
          copy S[h+1] through S[n] to V[1] through V[m];
          mergesort(h, U);
          mergesort(m, V);
          merge (h, m, U, V, S);
     }
}
```



Algorithm 2.3: Merge

Problem: Merge two sorted arrays into one sorted array.

Inputs: positive integers *h* and *m*, array of sorted keys *U* indexed from 1 to *h*, array of sorted keys *V* indexed from 1 to *m*.

Outputs: an array *S* indexed from 1 to *h* + *m* containing the keys in *U* and *V* in a single sorted array.

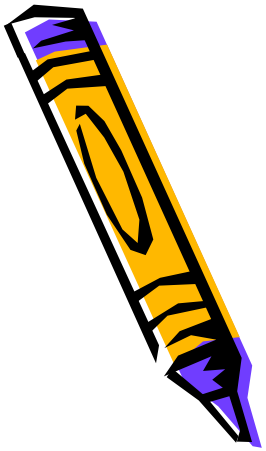
```
void merge (int h, int m, const keytype U[],
                          const keytype V[],
                          keytype S[])
{
  index i, j, k;

  i = 1; j = 1; k = 1;
  while (i <= h && j <= m) {
     if (U[i] < V[j]) {
            S[k] = U[i];
            i++;
       }
       else{
            S[k] = V[j];
            j++;
       }
       k++;
       }
  if (i>h)
        copy V[j] through V[m] to S[k] through S[h+m];
  else
        copy U[i] through U[h] to S[k] through S[h+m];
}
```

Table 2.1: An example of merging two arrays *U* and *V* into one array *S*[*]

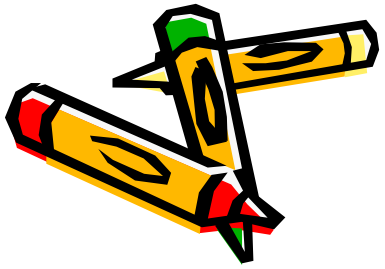
k	*U*	*V*	*S* (Resutl)
1	**10** 12 20 27	**13** 15 22 25	10
2	10 **12** 20 27	**13** 15 22 25	10 12
3	10 12 **20** 27	**13** 15 22 25	10 12 13
4	10 12 **20** 27	13 **15** 22 25	10 12 13 15
5	10 12 **20** 27	13 15 **22** 25	10 12 13 15 20
6	10 12 20 **27**	13 15 **22** 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 **25**	10 12 13 15 20 22 25
—	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27 ← Final values

[*]Items compared are in boldface. Items just exchanged appear in squares.

Analysis of Algorithm 2.2 Worst-Case Time Complexity (Mergesort)



- Basic operation:
  - the comparison of *U[i]* with *V[j]*.
  - As mentioned in Section 1.3, in the case of algorithms that sort by comparing keys, the comparison instruction and the assignment instruction can each be considered the basic operation. Here we will consider the comparison instruction. When we discuss Mergesort further in Chapter 7, we will consider the number of assignments.
- Input size: *h* and *m*
  - the number of items in each of the two input arrays.
- The worst case occurs when the loop is exited, because one of the indices—say, *i*—has reached its exit point *h*+1 whereas the other index *j* has reached *m*, 1 less than its exit point.



$$W(h, m) = h + m - 1.$$

Analysis of Algorithm 2.2 Worst-Case Time Complexity (Mergesort)

Basic operation: the comparison that takes place in *merge*.

Input size: *n*, the number of items in the array *S*.

The total number of comparisons is the sum of the number of comparisons in the recursive call to *mergesort* with *U* as the input, the number of comparisons in the recursive call to *mergesort* with *V* as the input, and the number of comparisons in the top-level call to *merge*. Therefore,

$$W(n) = \underbrace{W(h)}_{\text{Time to sort } U} + \underbrace{W(m)}_{\text{Time to sort } V} + \underbrace{h+m-1}_{\text{Time to merge}}$$

We first analyze the case where *n* is a power of 2. In this case,

$$h = \lfloor n/2 \rfloor = \frac{n}{2}$$

$$m = n - h = n - \frac{n}{2} = \frac{n}{2}$$

$$h + m = \frac{n}{2} + \frac{n}{2} = n.$$

Our expression for *W* (*n*) becomes

$$W(n) = W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + n - 1$$

$$= 2W\left(\frac{n}{2}\right) + n - 1.$$

Analysis of Algorithm 2.2 Worst-Case Time Complexity (Mergesort)

When the input size is 1, the terminal condition is met and no merging is done. Therefore, $W(1)$ is 0. We have established the recurrence

$$W(n) = 2W\left(\frac{n}{2}\right) + n - 1 \qquad \text{for } n > 1,\ n \text{ a power of } 2$$
$$W(1) = 0$$

This recurrence is solved in Example B.19 in Appendix B. The solution is

$$W(n) = n \lg n - (n-1) \in \Theta(n \lg n).$$

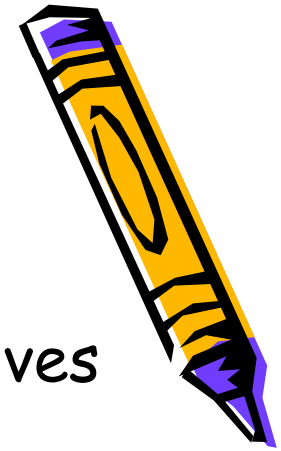
For n not a power of 2, we will establish in the exercises that

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1,$$

where $\lfloor y \rfloor$ and $\lceil y \rceil$ are the smallest integer $\geq y$ and the largest integer $\leq y$, respectively. It is hard to analyze this case exactly because of the floors ($\lfloor\rfloor$) and ceilings ($\lceil\rceil$). However, using an induction argument like the one in Example B.25 in Appendix B, it can be shown that $W(n)$ is nondecreasing. Therefore, Theorem B.4 in that appendix implies that

$$W(n) \in \Theta(n \lg n).$$

2.3 The Divide-and-Conquer Approach



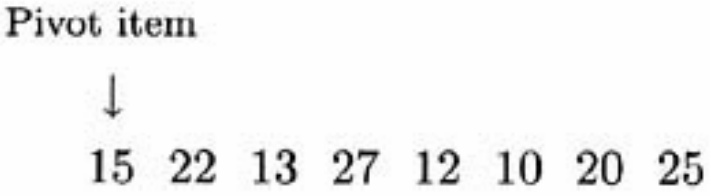
- The ***divide-and-conquer*** design strategy involves the following steps:
  - *(1) Divide* an instance of a problem into one or more smaller instances.
  - *(2) Conquer* (solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.
  - *(3) Combine,* if necessary, the solutions to the smaller instances to obtain the solution to the original instance.
- The reason we say "if necessary" in Step 3 is that in algorithms such as Binary Search Recursive (Algorithm 2.1) the instance is reduced to just one smaller instance, so there is no need to combine solutions

2.4 Quicksort (Partition Exchange Sort)

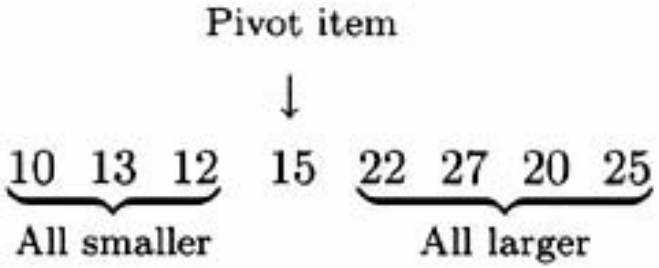
- Quicksort is similar to Mergesort in that the sort is accomplished by dividing the array into two partitions and then sorting each partition recursively.
- In Quicksort, however, the array is partitioned by placing all items smaller than some pivot item before that item and all items larger than the pivot item after it.
- The pivot item can be any item, and for convenience we will simply make it the first one.

Example 2.3

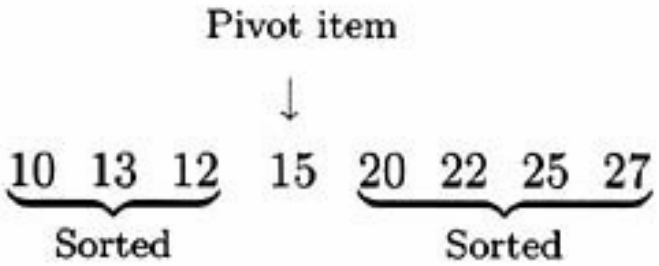
Suppose the array contains these numbers in sequence:



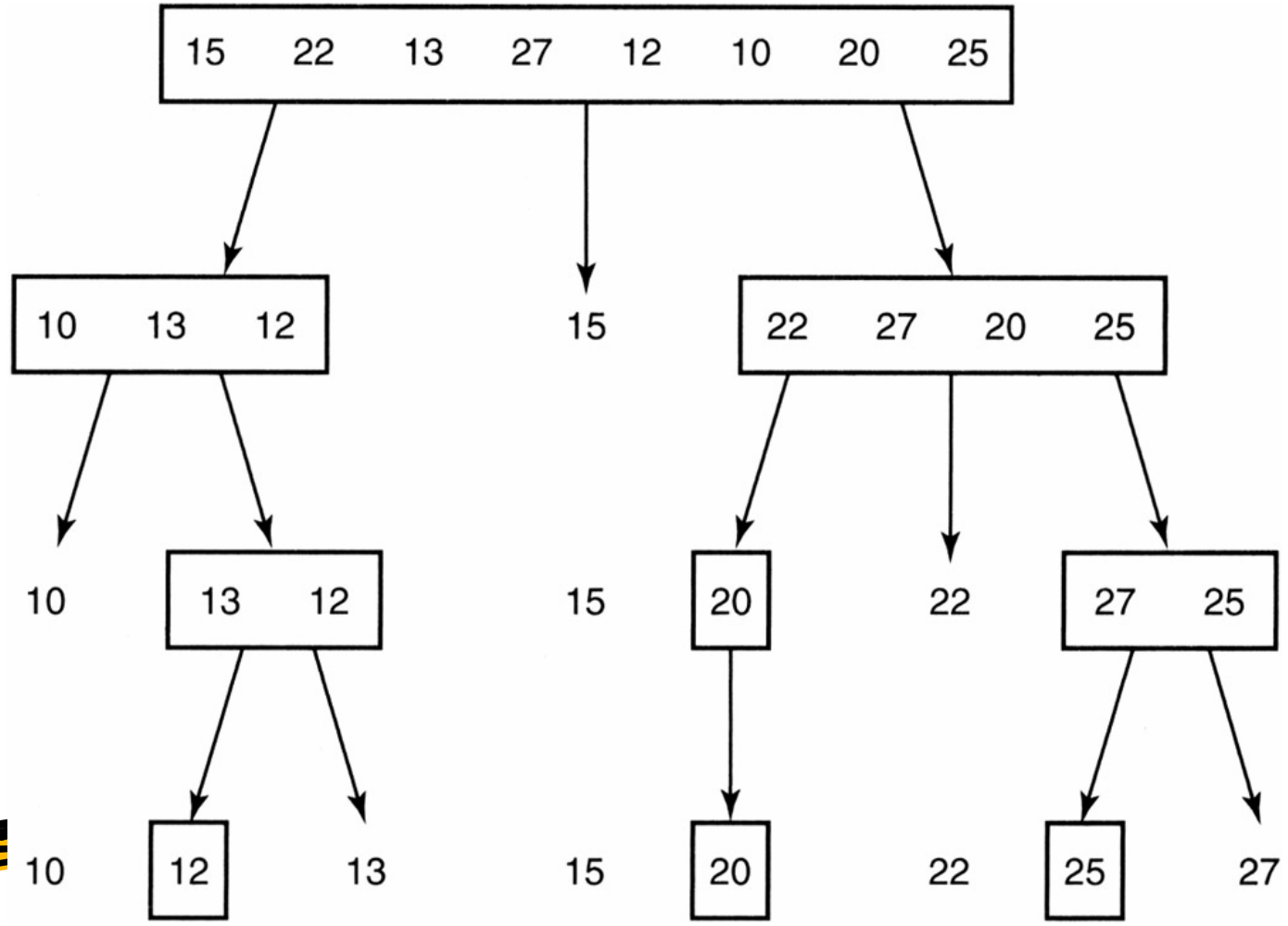
1. Partition the array so that all items smaller than the pivot item are to the left of it and all items larger are to the right:



2. Sort the subarrays



Example of QuickSort



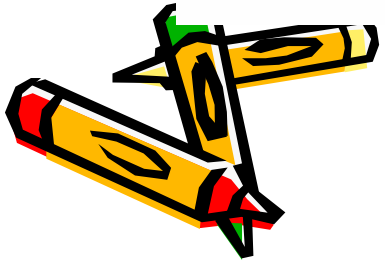
Algorithm 2.6: Quicksort

Problem: Sort *n* keys in nondecreasing order.

Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n*.

Outputs: the array *S* containing the keys in nondecreasing order.

```
void quicksort (index low, index high)
{
    index pivotpoint;

    if (high > low){
       partition(low, high, pivotpoint);
       quicksort(low, pivotpoint - 1);
       quicksort(pivotpoint + 1, high);
   }
}
```



Algorithm 2.7: Partition

Problem: Partition the array *S* for Quicksort.

Inputs: two indices, *low* and *high*, and the subarray of *S* indexed from *low* to *high*.

Outputs: *pivotpoint*, the pivot point for the subarray indexed from *low* to *high*.

```
void partition (index low, index high,
                                index& pivotpoint)
   {
     index i, j;
     keytype pivotitem;

     pivotitem = S[low];      // Choose first item for
     j = low;      //pivotitem.
     for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem){
                j++;
                exchange S[i] and S[j];
        }
     pivotpoint = j;
     exchange S[low] and S[pivotpoint];
}
```

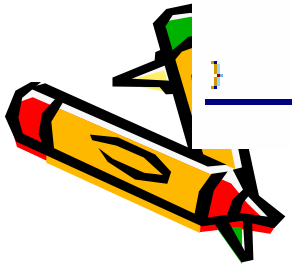


Table 2.2: An example of procedure *partition*[*]

i	*j*	*S*[1]	*S*[2]	*S*[3]	*S*[4]	*S*[5]	*S*[6]	*S*[7]	*S*[8]	
—	—	15	22	13	27	12	10	20	25	← Initial values
2	1	**15**	**22**	13	27	12	10	20	25	
3	2	**15**	22	**13**	27	12	10	20	25	
4	2	**15**	13	22	**27**	12	10	20	25	
5	3	**15**	13	22	27	**12**	10	20	25	
6	4	**15**	13	12	27	22	**10**	20	25	
7	4	**15**	13	12	10	22	27	**20**	25	
8	4	**15**	13	12	10	22	27	20	**25**	
—	4	10	13	12	15	22	27	20	25	← Final values

[*]Items compared are in boldface. Items just exchanged appear in squares.

Analysis of Algorithm 2.7 Every-Case Time Complexity (Partition)

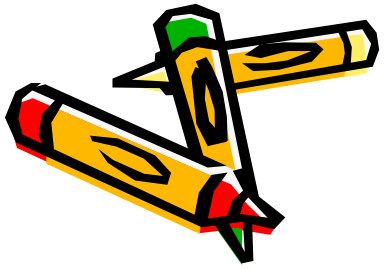
Basic operation: the comparison of *S[i]* with *pivotitem*.

Input size: $n = high - low + 1$, the number of items in the subarray.

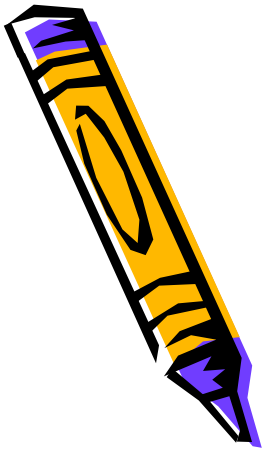
Because every item except the first is compared,

$T(n) = n - 1$.

We are using n here to represent the size of the subarray, not the size of the array S. It represents the size of S only when *partition* is called at the top level.

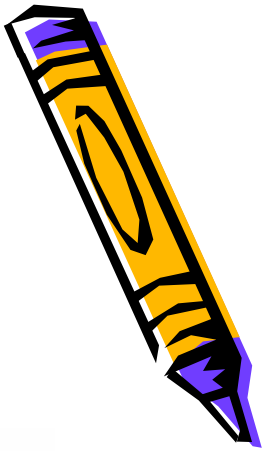


Analysis of Algorithm 2.6 Worst-Case Time Complexity (Quicksort)



- Basic operation:
  - the comparison of *S[i]* with *pivotitem* in *partition*.
- Input size:
  - *n*, the number of items in the array *S*.

Analysis of Algorithm 2.6 Worst-Case Time Complexity (Quicksort)



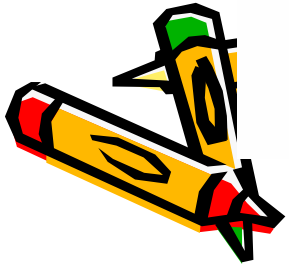
$$T(n) = \underbrace{T(0)}_{\text{Time to sort left subarray}} + \underbrace{T(n-1)}_{\text{Time to sort right subarray}} + \underbrace{n-1}_{\text{Time to partition}}$$

We are using the notation *T(n)* because we are presently determining the every-case complexity for the class of instances that are already sorted in nondecreasing order. Because *T*(0) = 0, we have the recurrence

$$\begin{aligned} T(n) &= T(n-1) + n - 1 \qquad \text{for } n > 0 \\ T(0) &= 0. \end{aligned}$$

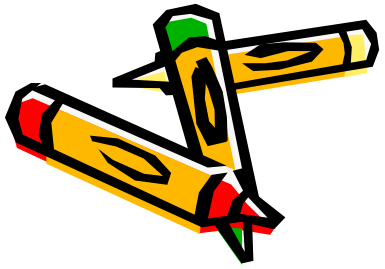
This recurrence is solved in Example B.16 in Appendix B. The solution is

$$T(n) = \frac{n(n-1)}{2}.$$

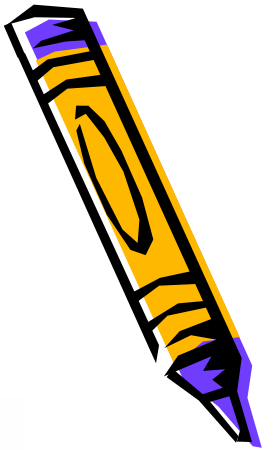


Why QuickSort is Quick?

- Wost-case time complexity
  - $O(n^2)$
  - *A slow algorithm*
- Average-case time complexity
  - $\Theta(n\lg n)$
  - *A quick algorithm*



Analysis of Algorithm 2.6 Average-Case Time Complexity (Quicksort)

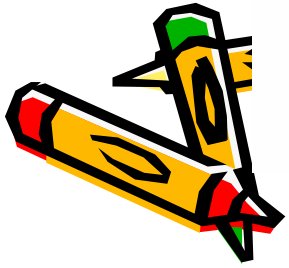


Basic operation: the comparison of *S[i]* with *pivotitem* in *partition*

Input size: *n*, the number of items in the array *S*.

We will assume that we have no reason to believe that the numbers in the array are in any particular order, and therefore that the value of *pivotpoint* returned by *partition* is equally likely to be any of the numbers from 1 through *n*. If there was reason to believe a different distribution, this analysis would not be applicable. The average obtained is, therefore, the average sorting time when every possible ordering is sorted the same number of times. In this case, the average-case time complexity is given by the following recurrence:

$$A(n) = \sum_{p=1}^{n} \overbrace{\frac{1}{n}}^{\substack{\text{Probability} \\ \textit{pivotpoint} \text{ is } p}} \underbrace{[A(p-1) + A(n-p)]}_{\substack{\text{Average time to} \\ \text{sort subarrays when} \\ \textit{pivotpoint} \text{ is } p}} + \underbrace{n-1}_{\substack{\text{Time to} \\ \text{partition}}} \qquad (2.1)$$



In the exercises we show that

$$\sum_{p=1}^{n} [A(p-1) + A(n-p)] = 2\sum_{p=1}^{n} A(p-1).$$

Plugging this equality into Equality 2.1 yields

$$A(n) = \frac{2}{n}\sum_{p=1}^{n} A(p-1) + n - 1.$$

Multiplying by n we have

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1). \qquad (2.2)$$

Applying Equality 2.2 to $n - 1$ gives

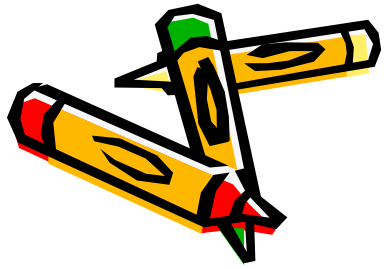
$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2). \qquad (2.3)$$

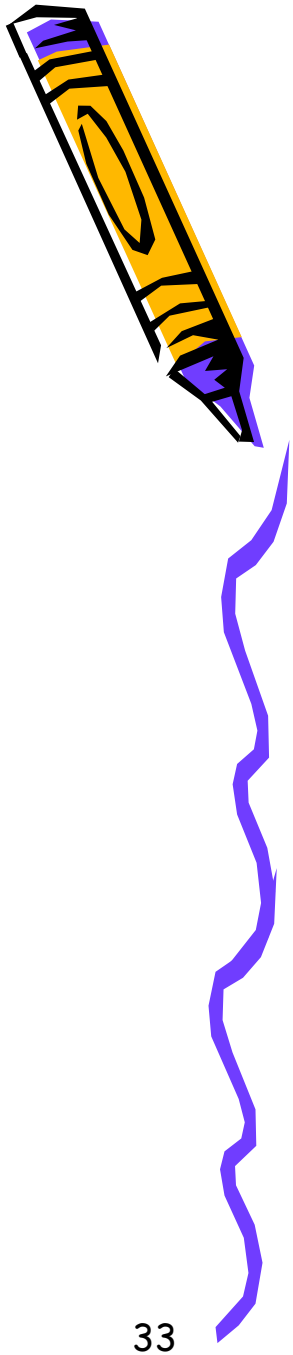
Subtracting Equality 2.3 from Equality 2.2 yields

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1),$$

which simplifies to

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}.$$





If we let

$$a_n = \frac{A(n)}{n+1},$$

we have the recurrence

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \qquad \text{for } n > 0$$

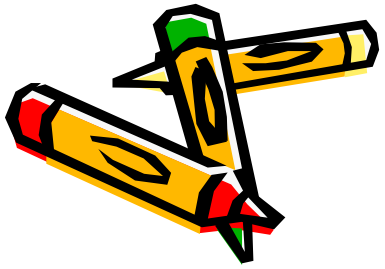
$$a_0 = 0.$$

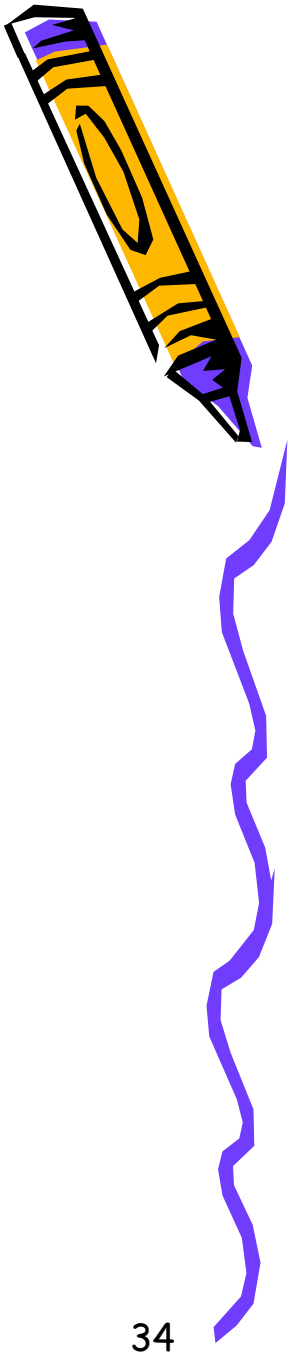
Like the recurrence in Example B.22 in Appendix B, the approximate solution to this recurrence is given by

$$a_n \approx 2\ln n,$$

which implies that

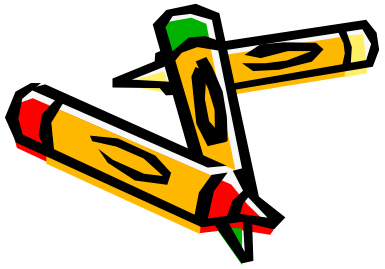
$$\begin{aligned} A(n) &\approx (n+1)\,2\ln n = (n+1)\,2\,(\ln 2)\,(\lg n) \\ &\approx 1.38\,(n+1)\lg n \in \Theta(n\lg n). \end{aligned}$$





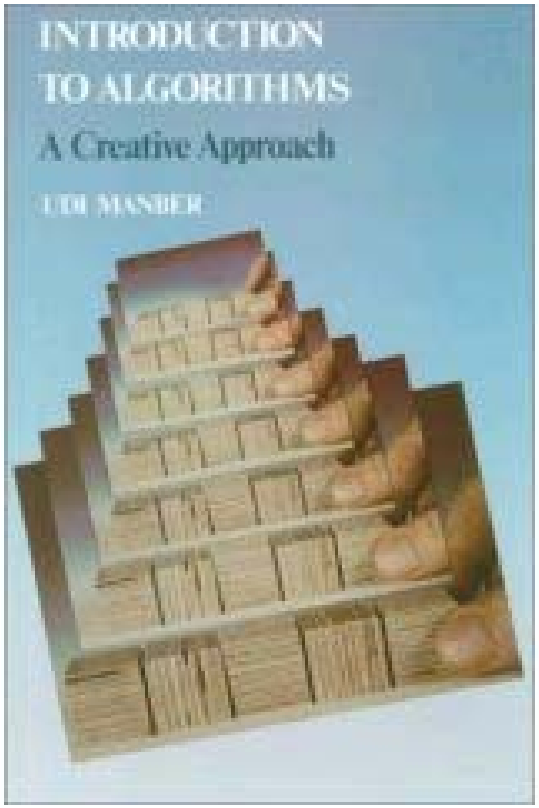
2.8 When Not to Use Divide-and-Conquer

- **Case I:** An instance of size *n* is divided into two or more instances each almost of size *n*.
  - Concept: like Napoleon dividing an opposing army of 30,000 soldiers into two armies of 29,999 soldiers (if this were somehow possible).
  - Example: Algorithm 1.6 (*n*th Fibonacci Term, Recursive)
    - $F(n)=F(n-1)+F(n-2) \Rightarrow T(n) > 2^{n/2}$
- **Case II:** An instance of size *n* is divided into almost *n* instances of size *n/c*, where *c* is a constant.
  - Complexity: $n^{\Theta(\lg n)}$



補充資料

- Divide-and-Conquer 使用時機



3.5.2 Divide and Conquer Relations

In a divide-and-conquer algorithm, the problem is divided into smaller subproblems, each subproblem is solved recursively, and a *combine* algorithm is used to solve the original problem. Assume that there are a subproblems, each of size $1/b$ of the original problem, and that the algorithm used to combine the solutions of the subproblems runs in time cn^k, for some constants a, b, c, and k. The running time $T(n)$ of the algorithm thus satisfies

$$T(n) = aT(n/b) + cn^k. \tag{3.14}$$

We assume, for simplicity, that $n = b^m$, so that n/b is always an integer (b is an integer greater than 1). We first try to expand (3.14) a couple of times to get the feel of it:

$$T(n) = a(aT(n/b^2) + c(n/b)^k) + cn^k = a(a(aT(n/b^3) + c(n/b^2)^k) + c(n/b)^k) + cn^k.$$

In general, if we expand all the way to $n/b^m = 1$, we get

$$T(n) = a(a(\cdots T(n/b^m) + c(n/b^{m-1})^k) + \cdots) + cn^k.$$

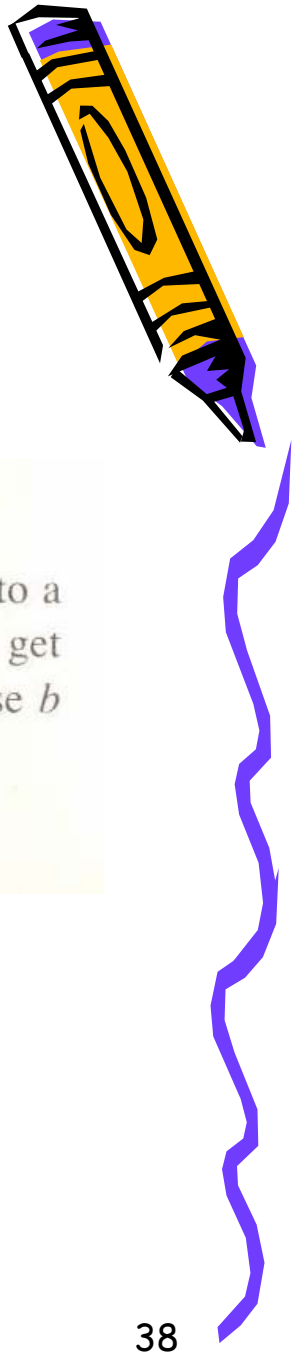
Let's assume that $T(1) = c$ (a different value would change the end result by only a constant). Then,

$$T(n) = ca^m + ca^{m-1}b^k + ca^{m-2}b^{2k} + \cdots + cb^{mk},$$

which implies that

$$T(n) = c\sum_{i=0}^{m} a^{m-i}b^{ik} = ca^m \sum_{i=0}^{m} \left(\frac{b^k}{a}\right)^i.$$

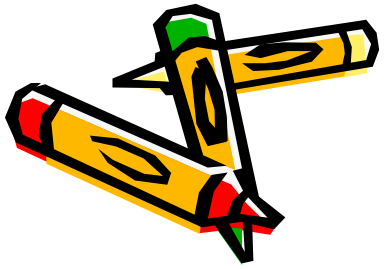
But, this is a simple geometric series. There are three cases, depending on whether (b^k/a) is less than, greater than, or equal to 1.

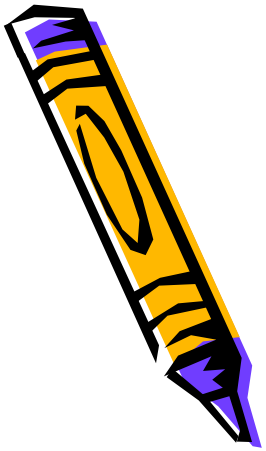


Case 1: $a > b^k$

In this case, the factor of the geometric series is less than 1, so the series converges to a constant even if m goes to infinity. Therefore, $T(n) = O(a^m)$. Since $m = \log_b n$, we get $a^m = a^{\log_b n} = n^{\log_b a}$ (the last equality can be easily proven by taking logarithm of base b of both sides). Thus,

$$T(n) = O(n^{\log_b a}).$$

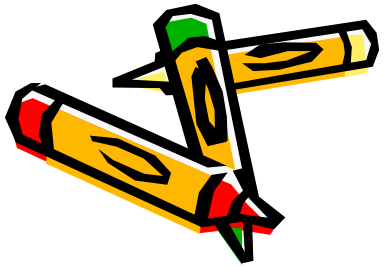


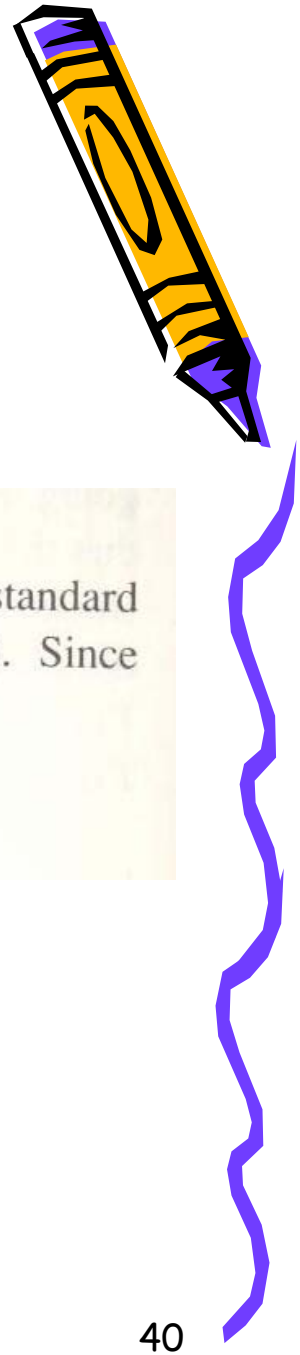


Case 2: $a = b^k$

In this case, the factor of the geometric series is 1, and thus $T(n) = O(a^m m)$. Notice that $a = b^k$ implies that $\log_b a = k$ and $m = O(\log n)$. Thus,

$$T(n) = O(n^k \log n).$$

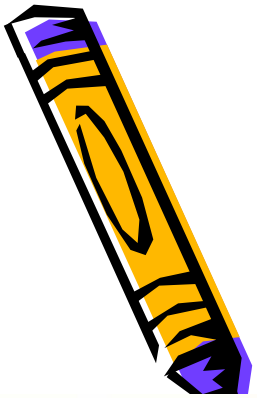




Case 3: $a < b^k$

In this case, the factor of the geometric series is greater than 1. We use the standard expression for summing a geometric series. Denote b^k/a by F (F is a constant). Since the first element of the series is a^m, we obtain

$$T(n) = a^m \frac{F^{m+1} - 1}{F - 1} = O(a^m F^m) = O((b^k)^m) = O((b^m)^k) = O(n^k).$$



These three cases are summarized in the following theorem.

□ Theorem 3.4

The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, *where* a *and* b *are integer constants,* $a \geq 1$, $b \geq 2$, *and* c *and* k *are positive constants, is*

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