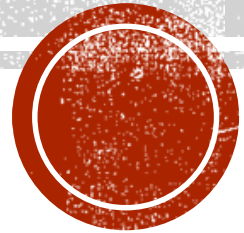
Lect 02. Divide and Conquer



Spring, 2020

School of Computer Science & Engineering

Pusan National University

Divide and conquer

Binary Search with Divide and Conquer

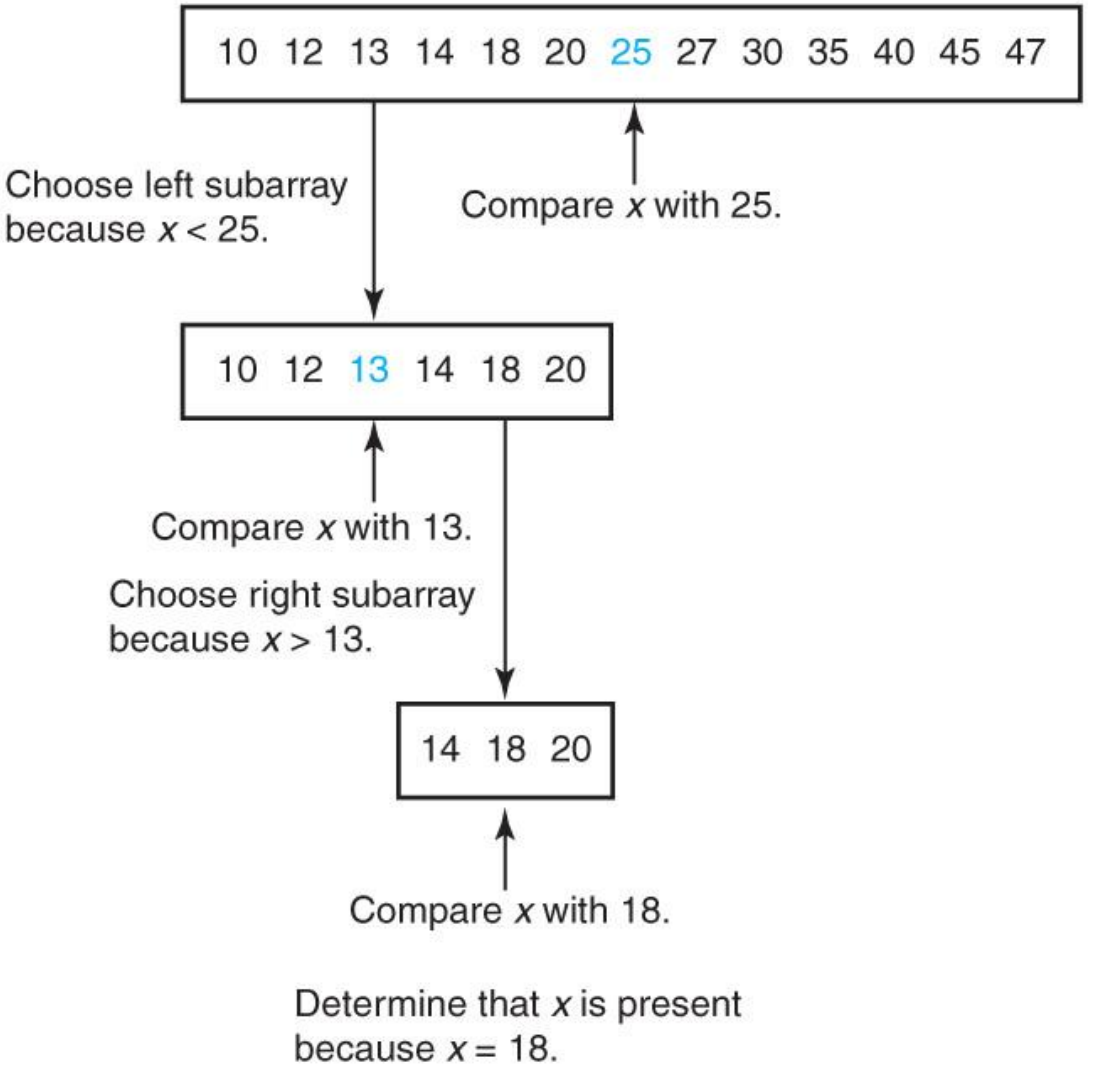


Figure 2.1 : The steps down by a human when searching with Binary Search. (Note: x = 18)

Divide-and-Conquer approach

- Step 1: Divide an instance of a problem into one or more smaller instances.
- Step 2: Conquer solve each of the smaller instances(use recursion until the array is sufficiently small).
- Step 3: Combine the solutions of the smaller instances to obtain the solution of the original instance.

-> Top-down approach

Binary Search : Recursive Algorithm

- Problem: Determine whether *x* is in the sorted array *S* of size *n*.
- Input : positive integer *n*, sorted(nondecreasing order) array of keys *S* indexed from 1 to *n*, a key *x*
- Output : *location,* the location of *x* in *S* (0 if *x* is not in *S*).
- method:
  - if x == array[middle], then "Find" else :
  - **divide**: divide the array and check the value of array[middle]. If the value >x then select left of array, else select right of array
  - **conquer**: find x in selected half array.
  - **combine**: (if need)

Binary Search : Recursive Algorithm

```
index location (index low, index high) {
  index mid;

  if (low > high)
      return 0;                  // fail
  else {
      mid = (low + high) / 2           // integer division()
      if (x == S[mid])
        return mid;                     // find
      else if (x < S[mid])
        return location(low, mid-1);    // select left side
      else
        return location(mid+1, high);  // select right side
  }
}
…
locationout = location(1, n);
...
```

Example

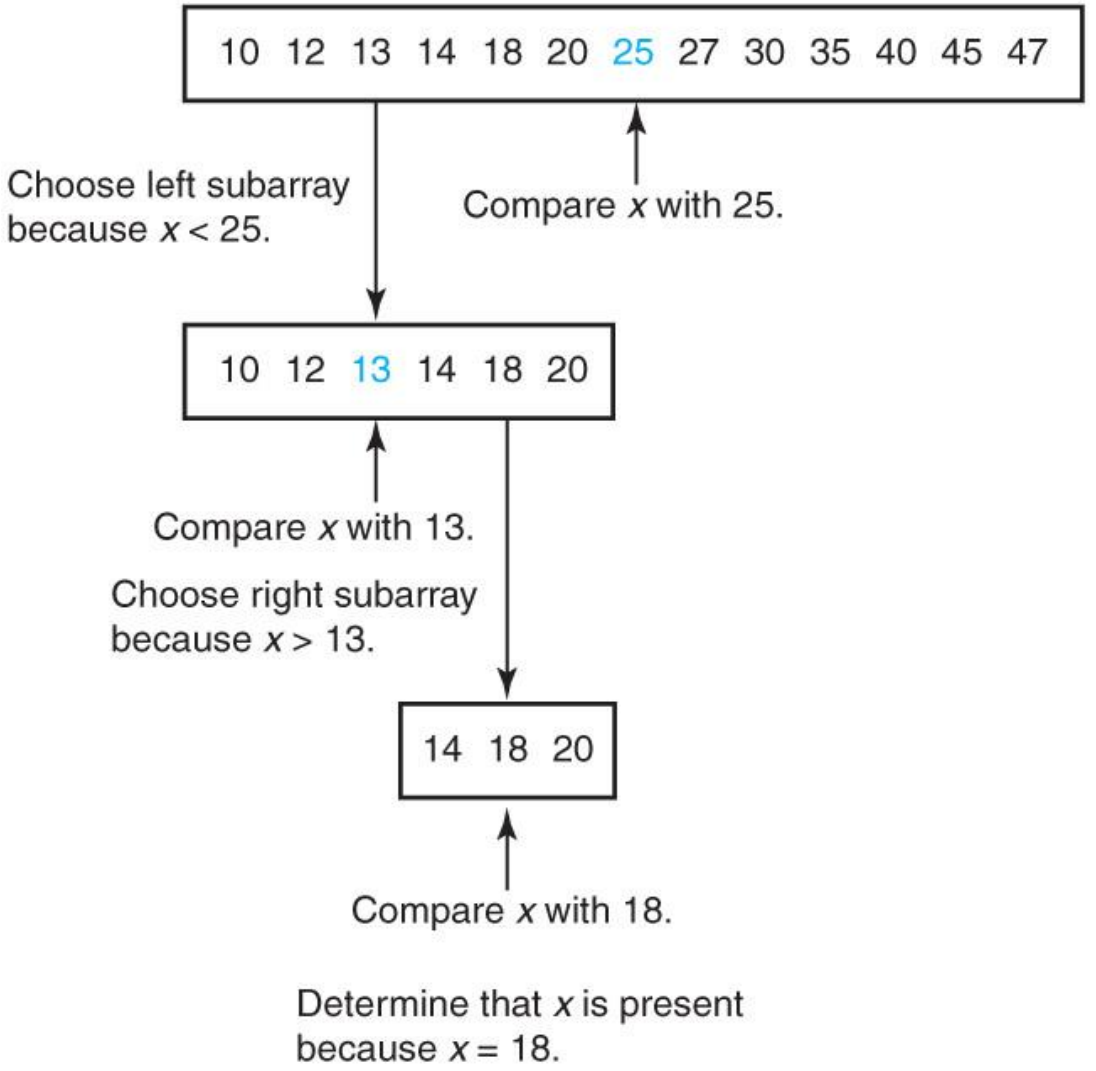


Figure 2.1 : The steps down by a human when searching with Binary Search. (Note: x = 18)

Discussion

- Binary Search is the simplest kind of divide-and-conquer algorithm because the instance is broken down into only one smaller instance, so there is no combination of outputs. The solution of the original instance is the solution to the smaller instance.
- This recursive version of Binary Search employs that no operations are done after the recursive call (**tail-recursion**); it is straightforward to produce an iterative version.
- Indeed it is better to use iteration since iterative algorithms **execute faster** than recursive algorithms because **no stack** needs to be maintained.

Worst-Case Time complexity

- Basic operation: the comparison of *x* with S[mid]
- Input size: n, the number of items in the array (= *high* - *low* + 1)

Worst-Case Time Complexity

$\frac{n}{2}$

- **Case 1: n is a power of 2.**

  we have the following recurrence:

$$W(n) = W\left(\frac{n}{2}\right) + 1,\ n > 1 \text{ and } n = 2^k (k \geq 1)$$

$$W(1)=1$$

The solution of the expression is the followings:

$$W(1) = 1$$
$$W(2) = W(1) + 1 = 2$$
$$W(4) = W(2) + 1 = 3$$
$$W(8) = W(4) + 1 = 4$$
$$W(16) = W(8) + 1 = 5$$
$$W(2^k) = k + 1..$$
$$W(n) = \lg n + 1$$

Worst-Case Time Complexity

Proof: mathematical Induction:

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

ii) assume- positive integer n is a power of 2, $W(n) = \lg n + 1$

iii) Induction: show that $W(2n) = \lg(2n) + 1$.

Using the recurrence,

$$
\begin{aligned}
W(2n) &= W(n) + 1 && \textit{by recurrence} \\
&= \lg n + 1 + 1 && \textit{by induction} \\
&= \lg n + \lg 2 + 1 \\
&= \lg(2n) + 1
\end{aligned}
$$

Worst-Case Time Complexity

- **Case 2: in general – the size of half array is** $\left\lfloor \frac{n}{2} \right\rfloor$
- **If** n **is a middle of index** $mid = \left\lfloor \frac{1+n}{2} \right\rfloor$, **the size of sub array is the followings respectively.**

n	Size of left sub-array	mid	Size of right sub-array
Even	n/2 - 1	1	n/2
odd	(n-1)/2	1	(n-1)/2

By the above table, counter of item to find next step is at least $\left\lfloor \frac{n}{2} \right\rfloor$. Therefore the recurrence is the following:

$$W(n) = 1 + W(\left\lfloor \tfrac{n}{2} \right\rfloor) \quad n > 1$$

$$W(1) = 1$$

Worst-Case Time Complexity

W(n)=W(n/2)+1
W(n)=lg n + 1

- $W(n)=\lfloor \lg n \rfloor+1$ prof by mathematics induction for all *n*.

i) *n* = 1, true. $\lfloor \lg n \rfloor+1=\lfloor \lg 1 \rfloor+1=0+1=1=W(1)$

ii) *n* > 1, for all k (1 < *k* < *n*), we assume that $W(k)=\lfloor \lg k \rfloor+1$

iii) induction step - n : even($\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$)

),

$$
\begin{aligned}
W(n) &= 1+W(\lfloor \tfrac{n}{2} \rfloor) && \textit{by recurrence} \\
&= 1+\lfloor \lg \lfloor \tfrac{n}{2} \rfloor \rfloor+1 && \textit{by induction} \\
&= 2+\lfloor \lg \lfloor \tfrac{n}{2} \rfloor \rfloor \\
&= 2+\lfloor \lg \tfrac{n}{2} \rfloor && n : \textit{even} \\
&= 2+\lfloor \lg n-1 \rfloor \\
&= 2+\lfloor \lg n \rfloor-1 \\
&= 1+\lfloor \lg n \rfloor
\end{aligned}
$$

Worst-Case Time Complexity

- n: odd($\left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2}$),

$$
\begin{aligned}
W(n) &= 1 + W\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) && \text{by recurrence} \\
&= 1 + \left\lfloor \lg \left\lfloor \tfrac{n}{2} \right\rfloor \right\rfloor + 1 && \text{by induction} \\
&= 2 + \left\lfloor \lg \left\lfloor \tfrac{n}{2} \right\rfloor \right\rfloor \\
&= 2 + \left\lfloor \lg \tfrac{n-1}{2} \right\rfloor && n \text{ odd} \\
&= 2 + \left\lfloor \lg (n-1) - 1 \right\rfloor \\
&= 2 + \left\lfloor \lg (n-1) \right\rfloor - 1 \\
&= 1 + \left\lfloor \lg (n-1) \right\rfloor \\
&= 1 + \left\lfloor \lg n \right\rfloor && n : \text{odd}
\end{aligned}
$$

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

Merge Sort

Merge Sort

- Using **two-way merging**, we can combine two sorted arrays into one array. By repeatedly applying the merging procedure, we can sort an array.
- Eventually the size of the subarrays will become 1, and an array of size 1 is trivially sorted.
- This procedure is called **mergesort**.

Merge Sort

- Mergesort has three steps.
  - Step 1: Divide the array into 2 sub-arrays each of n/2.
  - Step 2: Solve each sub-array by sorting it (use recursion till array is sufficiently small).
  - Step 3: Combine solutions to the sub-arrays by merging them into a single sorted array.

Mergesort

- Problem: Sort *n* keys in nondecreasing sequence.
- Inputs: positive integer *n*, array of keys S[1..n]
- Outputs: the array S[1..n] containing the keys in nondescreasing order.
- Example : 27, 10, 12, 20, 25, 13, 15, 22

Merge Sort

- Algorithm

```
void mergesort (int n, keytype S[]) {
        const int h = n / 2, m = n - h;
        keytype U[1..h], V[1..m];

        if (n > 1) {
            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);
        }
}
```

Merge

- Problem: Merged two sorted arrays into one sorted.
- Inputs: (1)positive integers *h* and *m*, (2)array of sorted keys U[1..h], V[1..m]
- Outputs: an array S[1…h+m] containing the keys in U and V in a single sorted array.

- **Algorithm:**

```
void merge(int h, int m, const keytype U[], const keytype V[], const 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];
}
```

Example

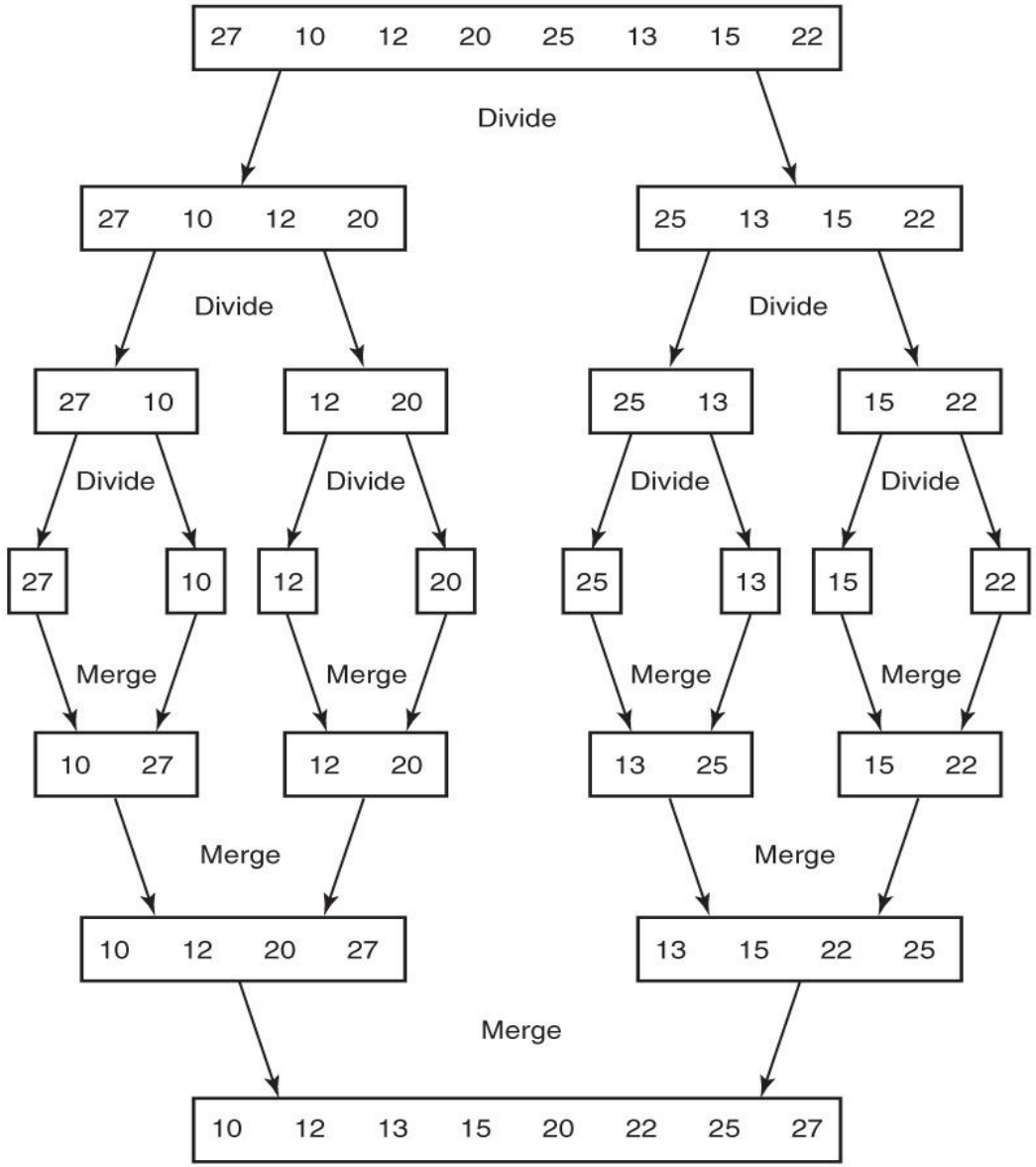


Figure 2.2: The steps done by a human when sorting with Mergesort.

Example - merge

- Merging two arrays U and V into one array S

k	U	V	S(result)
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)

Merge Sort

- In Mergesort, we are concentrating on comparison. Suppose we have n keys we will not have more than n comparisons. In the worst case we will have n–1 and in the normal case we will have $C \leq n \log(n+1)$. $C = O(n \log n)$.

- An **in-place sort** is a sorting algorithm that does not use any extra space beyond that needed to store the input.

Analysis of Time Complexity

- 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.
  - Analysis:
    - The total # of comparisons
      - The # of comp. in the recursive call to *mergesort* with U + the # of comp. in the recursive call to *mergesort* with V + the # of comp. in the top-level call to *merge*.
    - W(n) = W(h) + W(m) + h+m-1

      Time to sort U Time to sort V Time to merge

Analysis of Time Complexity

- Case n : power of 2
  - h=$\lfloor n/2 \rfloor = \frac{n}{2}$
  - $m = n - h = n - \frac{n}{2} = \frac{n}{2}$
  - $h + m = \frac{n}{2} + \frac{n}{2} = n.$
  - $\therefore W(n) = W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + n - 1$
  - $= 2W\left(\frac{n}{2}\right) + n - 1.$
  - When the input size is 1, the terminal condition is met and no merging is done. $\therefore W(1) = 0.$

Analysis of Time Complexity

- The recurrence
- $W(n) = 2W\left(\frac{n}{2}\right) + n - 1, for\ n > 1, n = 2^k$
- $W(1) = 0$
- Let $n = 2^k, T(2^k) = 2T\left(\frac{2^k}{2}\right) + 2^k - 1$
- $= 2T(2^{k-1}) + 2^k - 1.$
- Let $t_k = T(2^k), t_k = 2t_{k-1} + 2^k - 1.$
- Apply the theorem(nonhomogeneous linear recurrence), $t_k = c_1 + c_2 2^k + c_3 k 2^k.$
- $T(2^k) = c_1 + c_2 n + c_3 n \lg n.$
- $T(n) = c_1 + c_2 n + c_3 n \lg n.$
- $T(1) = c_1 + c_2 = 0, T(2) = c_1 + 2c_2 + c_3 2 \lg 2 = 1.$
- Let $c_3 = 1$, the constants are $c_1 = 1, c_2 = -1$

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

Analysis of Time Complexity

- Case $n \neq 2^k$,
  - $W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$
  - $W(n) \in \Theta(n \lg n)$.

Advanced Mergesort(space complexity)

- Mergesort2
  - Problem: Sort n keys in nondecreasing sequence.
  - Inputs: positive integer n, array of keys S[1..n]
  - Outputs: the array S[1..n] containing the keys in nondecreasing order
  - Algorithm:

```
void mergesort2 (index low, index high) {
        index mid;
        if (low < high) {
            mid = (low + high) / 2;
            mergesort2(low, mid);
            mergesort2(mid+1, high);
            mergesort2(low, mid, high);
        }
}
  …
mergesort2(1, n);
  …
```

Advanced Merge

- Merge
  - **Problem**: Merge the two sorted subarrays of *S* created in mergesort 2
  - **Inputs**: (1) indices *low, mid, high*, (2) subarray of *S*[*low*..*high*], where *S*[*low*..*mid*] and *S*[*mid*+1..*high*] are already sorted.
  - **Outputs**: the subarray of *S* indexed from *low* to *high* containing the keys in nondecreasing order.

- Algorithm:

```
void merge2(index low, index mid, index high) {
        index i, j, k;   keytype U[low..high];   // a local array needed for the merging
        i = low; j = mid + 1; k = low;
        while (i <= mid && j <= high) {
            if (S[i] < S[j]) {
                U[k] = S[i];
                i++;
            }
            else {
                U[k] = S[j];
                j++;
            }
            k++;
        }
        if (i > mid)
            copy S[j] through S[high] to U[k] through U[high];
        else
            copy S[i] through S[mid] to U[k] through U[high];
        copy U[low] through U[high] to S[low] through S[high];
}
```

Quick Sort

Quicksort(Partition Exchange Sort)

- Developed by C.A.R. Hoare(UK, 1962)
- Example: 15 22 13 27 12 10 20 25

Ex

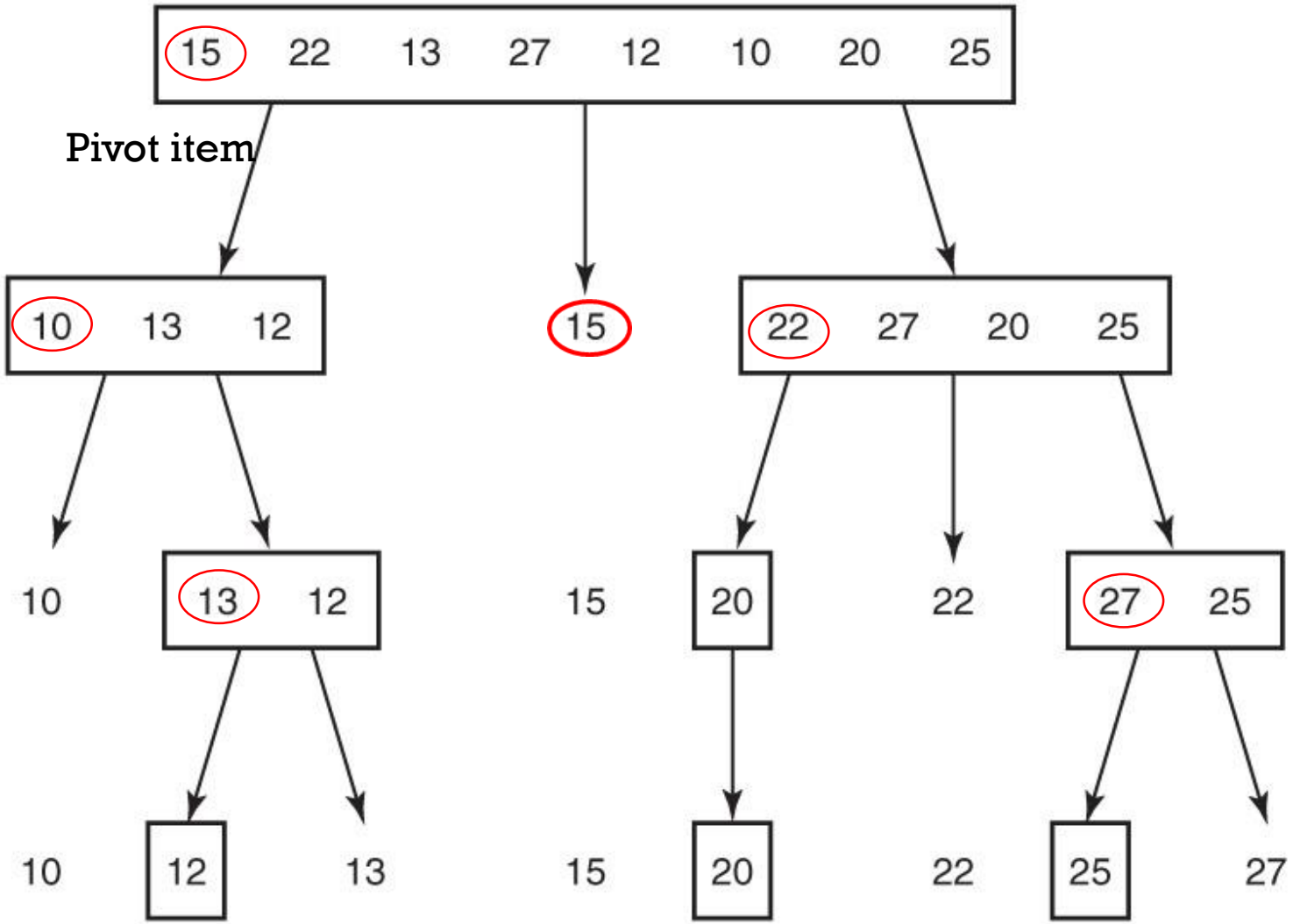


Figure 2.3: The steps done by a human when sorting with Quicksort. The subarrays are enclosed in rectangles whereas the pivot points are free.

Quicksort algorithm

- Problem: Sort n keys in nondecreasing order
- Inputs: positive integer n, array of keys S[1..n]
- Outputs: the array S[1..n] containing the keys in nondecreasing order
- Algorithm:

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

Partition algorithm

- Problem : Partition the array S for Quicksort
- Inputs: (1) two indices low,high, (2) the subarray S[low … high]
- Outputs: *pivotpoint*, the pivot point for the subarray indexed from *low* to *high*
- Algorithm:

```
void partition (index low, index high, index& pivotpoint) {
        index i, j;
        keytype pivotitem;
        pivotitem = S[low];        //Choose first item for pivotitem
        j = low;
        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];    // put pivotitem  at pivotpoint
}
```

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 value
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 Value

Analysis-partition

- Basic operation : the comparison of S[i] with pivotitem
- Input size : n=high-low+1, the # of items in the subarray
  - Compared to first item

    T(n)=n-1

Analysis-worst-case Time complexity

- Basic Op. : the comparison of S[i] with *pivotitem* in *partition*.
- Input size : n, the # of items in the array S.
- Analysis
  - Worst case : sorted items
  - $T(n) = T(0) + T(n-1) + n - 1$

    Time to sort left subarray — Time to sort right subarray — Time to partition

$$T(n) = T(n-1) + n - 1 \quad for\ n > 0$$

$$T(0) = 0$$

$$T(n) = T(n - 1) + n - 1$$

Analysis-worst-case Time complexity

- $T(n) = T(n - 1) + n - 1$

$$T(n - 1) = T(n - 2) + n - 2$$
$$T(n - 2) = T(n - 3) + n - 3$$
$$\cdots$$
$$T(2) = T(1) + 1$$
$$T(1) = T(0) + 0$$
$$T(0) = 0$$

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

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

- Show that, for all n, $W(n) \le \frac{n(n-1)}{2}$

  Proof: (mathematical induction)

  Induction base: $n = 0$, $W(0) \le \frac{0(0-1)}{2}$

  Induction hypothesis: Assume that for $0 \le k < n$, $W(k) \le \frac{k(k-1)}{2}$

  Induction step: show that $W(n) \le \frac{n(n-1)}{2}$

$$W(n) \le W(p-1) + W(n-p) + n - 1 \text{ by recurrence, pivotpoint} = p$$

$$\le \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1 \text{ By induction hypothesis}$$

$$= \frac{p^2 - 3p + 2 + (n-p)^2 - n + p + 2n - 2}{2}$$

$$= \frac{p^2 + (n-p)^2 + n - 2p}{2}$$

p= n – 1 : max, therefore $max_{1 \le p \le n-1}(p^2 + (n-p)^2) = 1^2 + (n-1)^2 = n^2 - 2n + 2$

$$W(n) \le \frac{p^2 + (n-p)^2 + n - 2p}{2} \le \frac{n^2 - 2n + 2 + n - 2}{2} = \frac{n^2 - n}{2} = \frac{n(n-1)}{2}$$

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Analysis – Average case

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

Probability pivotpoint is p

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

Average time to sort subarrays When *pivotpoint* is p

Time to partition

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

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

Multiplying by n, $nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1)$ (1)

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

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

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

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

We have the recurrence. $a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)}, \quad n > 0 \quad a_0 = 0$

Then, $a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \quad a_{n-1} = a_{n-2} + \frac{2(n-2)}{(n-1)n} \cdots \quad a_2 = a_1 + \frac{1}{3} \quad a_1 = a_0 + 0$

Therefore, the solution is

$$a_n = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)}$$
$$= 2\left(\sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}\right)$$

where right term is ignore because it's too small.

$\ln n = \log_e n$,

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n$$

The solution is $a_n \approx 2 \ln n$. Therefore, $a_n = \frac{A(n)}{n+1} \approx 2 \ln n$

$$A(n) \approx (n+1)2 \ln n$$
$$= (n+1)2(\ln 2)(\lg n)$$
$$\approx 1.38(n+1) \lg n$$
$$\in \Theta(n \lg n)$$

Motivation: Master Theorem-Asymptotic Behavior of Recursive Algorithms

- When analyzing algorithms, recall that we only care about the asymptotic behavior
- Recursive algorithms are no different
- Rather than solving exactly the recurrence relation associated with the cost of an algorithm, it is sufficient to give an asymptotic characterization
- The main tool for doing this is the master theorem

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

$$T(n) = a\,T(n/b) + f(n)$$

$$T(1) = c$$

where $a \geq 1, b \geq 2, c>0$. If f(n) is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Pitfalls

- You cannot use the Master Theorem if
  - T(n) is not monotone, e.g. $T(n) = sin(x)$
  - f(n) is not a polynomial, e.g., $T(n)=2T(n/2)+2^n$
  - b cannot be expressed as a constant, e.g.

$$T(n) = T(\sqrt{n})$$

- Note that the Master Theorem does not solve the recurrence equation

Master Theorem: Example 1

- Let $T(n) = T(n/2) + \frac{1}{2} n^2 + n$. What are the parameters?

a = 1

b =2

d =2

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Therefore, which condition applies? $f(n) = \Theta(n^2), d = 2$

$1 < 2^2$, case 1 applies

We conclude that $T(n) \in \Theta(n^d) = \Theta(n^2)$

Master Theorem: Example 2

- Let T(n)= 2 T(n/4) + √n + 42. What are the parameters?

  a =2

  b =4

  d =1/2

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Therefore, which condition applies?

$2 = 4^{1/2}$, case 2 applies

We conclude that

$$T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$$

Master Theorem: Example 3

- Let T(n)= 3 T(n/2) + 3/4n + 1. What are the parameters?

  a =3

  b =2

  d =1

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Therefore, which condition applies?

$3 > 2^1$, case 3 applies

We conclude that

$$T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$$

Note that $\log_2 3 \approx 1.584\ldots$, can we say that $T(n) \in \Theta(n^{1.584})$

No, because $\log_2 3 \approx 1.5849\ldots$ and $n^{1.584} \notin \Theta(n^{1.5849})$

Matrix Multiplication

- Standard Matrix Multiplication Algorithm
  - Problem: Determine the product of two $n \times n$ matrices.
  - Inputs: positive integer n, and two $n \times n$ matrices A and B.
  - Outputs: the product C of A and B.
  - Algorithm:

```
void matrixmult (int n, const number A[][], const number B[][], number C[][]) {
        index i, j, k;
        for (i = 1; i <= n; i++)
                for (j = 1; j <= n; j++) {
                        C[i][j] = 0;
                        for (k = 1; k <= n; k++)
                                C[i][j] = C[i][j] + A[i][k] * B[k][j];
                }
}
```

Matrix Multiplication

- Time Complexity I:
  - Basic Operation: multiplication instruction in the innermost for loop
  - Input size: n, the number of rows and columns.
  - Time complexity analysis of all case : the total number of multiplication is $T(n) = n \times n \times n = n^3 \in \Theta(n^3)$.

- Time Complexity II:
  - Basic Operation : addition instruction in the innermost for loop
  - Input size: n, the number of rows and columns
  - Time complexity analysis of all case : the total number of addition is $T(n) = (n-1) \times n \times n = n^3 - n^2 \in \Theta(n^3)$.

2×2 Matrix Multiplication: Strassen's Algorithm

- Problem: product *C* of two 2×2 matrix *A and B*

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

- Strassen's Solution:

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where

$$\begin{aligned} m_1 &= (a_{11} + a_{22}) \times (b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22}) \times b_{11} \\ m_3 &= a_{11} \times (b_{12} + b_{22}) \\ m_4 &= a_{22} \times (b_{21} + b_{11}) \\ m_5 &= (a_{11} + a_{12}) \times b_{22} \\ m_6 &= (a_{21} - a_{11}) \times (b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22}) \times (b_{21} + b_{22}) \end{aligned}$$

- Time Complexity Analysis :
  - Standard method: multiplications – 8 times and additions/subtractions – 4 times.
  - Strassen's method: multiplications- 7 and addition/subtractions- 18.
  - At glance, it is not very impressive but Strassen's method is good according to the increasing input size.

n × n Matrix Multiplication: Strassen's Algorithm

- Problem : Determine the product of two $n \times n$ matrices where n is a power of 2. The product C of A and B is obtained by combining the four submatrices.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

- Strassen's Solution :

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

where

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) \times B_{11} \\ M_3 &= A_{11} \times (B_{12} + B_{22}) \\ M_4 &= A_{22} \times (B_{21} + B_{11}) \\ M_5 &= (A_{11} + A_{12}) \times B_{22} \\ M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{aligned}$$

Strassen's Algorithm

- Problem: Determine the product of two $n \times n$ matrices where n is a power of 2.
- Inputs: an integer n that is a power of 2, and two $n \times n$ matrices A and B
- Outputs: the product C of A and B.
- Algorithm:

```
void strassen (int n, n*n_matrix A, n*n_matrix B, n*n_matrix& C) {
        if (n <= threshold )
            compute C = A x B using the standard algorithm ;
        else {
            partition A into four submatrices A11, A12, A21, A22 ;
            partition B into four submatrices B11, B12, B21, B22 ;
            compute C = A x B using Strassen's method ;
            // example recursive call: strassen(n/2, A11+A12, B11+B22,M1)
        }
}
```

- *threshold* : the point at which we feel it is more efficient to use the standard algorithm than it would be to call procedure *strassen* recursively.

Analysis

- Time Complexity I
  - Basic operation: one elementary multiplication
  - Input sizes: n, the number of rows and columns in the matrices
  - Analysis of time complexity in all cases: If threshold is 1, (threshold doesn't affect the order at all.)

    the recurrence is

$$T(n) = 7T(\tfrac{n}{2}) \quad n > 10 \lfloor \rfloor, \; n = 2^k (k \geq 1)$$
$$T(1) = 1$$

    we expend the recurrence,

$$\begin{aligned} T(n) &= 7 \times 7 \times \cdots \times 7 \quad (k \;\; times) \\ &= 7^{k} \\ &= 7^{\lg n} \\ &= n^{\lg 7} \\ &= n^{2.81} \\ &\in \Theta(n^{2.81}) \end{aligned}$$

    The result is able to verify by mathematical induction. This recurrence can be verified by master theorem.

Analysis

- Analysis of time complexity II
  - Basic operation: one elementary addition or subtraction
  - Input size: n, the # of rows and columns in the matrices
  - Analysis of time complexity in all cases: If threshold is 1, the recurrence is

$$T(n) = 7T(\tfrac{n}{2}) + 18(\tfrac{n}{2})^2 \quad n > 1 \text{ and } n = 2^k (k \geq 1)$$

$$T(1) = 0$$

By the master's theorem, it is solved easily the following:

$$T(n) = \Theta(n^{\lg_2 7}) = \Theta(n^{2.81})$$

Discussion

- There is no algorithm whose time complexity is $\Theta(n^2)$
- In addition, no one has proven that it is not possible to create such algorithm.

When Not to Use Divide-and-Conquer

- Divide-and-conquer algorithms should be avoided in the following two cases:
  - 1. An instance of size n is divided into **two or more instances each almost of size n**. => time complexity : exponential time
  - 2. An instance of size n is divided into **almost n instances of size n/c**, where c is a constant. => time complexity : exponential time $\Theta(n^{\lg n})$