Chapter 2
Divide-and-Conquer

Defeat Them in Detail: The Divide and Conquer Strategy. Look at the parts and determine how to control the individual parts, create dissension and leverage it.

—Robert Greene

The divide-and-conquer is an important technique for design of algorithms. In this chapter, we will employ several examples to introduce this technique, including the rectilinear minimum spanning tree, the Fibonacci search method, and the sorting problem. Sorting is not a combinatorial optimization problem. However, it appears in algorithms very often as a procedure, especially in algorithms for solving combinatorial optimization problems. Therefore, we would like to make more discussion in this chapter.

2.1 Algorithms with Self-Reducibility

There exist a large number of algorithms in which the problem is reduced to several subproblems, each of which is the same problem on a smaller-size input. Such a problem is said to have the self-reducibility, and the algorithm is said to be with self-reducibility.

For example, consider sorting problem again. Suppose input contains n numbers. We may divide these n numbers into two subproblems. One subproblem is the sorting problem on $\lfloor n/2 \rfloor$ numbers, and the other subproblem is the sorting problem on $\lceil n/2 \rceil$ numbers. After completely sorting each subproblem, combine two sorted sequences into one. This idea will result in a sorting algorithm, called the *merge sort*. The pseudocode of this algorithm is shown in Algorithm 1.

The main body calls a procedure. This procedure contains two self-calls, which means that the merge sort is a recursive algorithm, that is, the divide will continue until each subproblem has input of single number. Then this procedure employs another procedure (Merge) to combine solutions of subproblems with smaller

D.-Z. Du et al., *Introduction to Combinatorial Optimization*, Springer Optimization and Its Applications 196, https://doi.org/10.1007/978-3-031-10596-8_2

Algorithm 1 Merge sort

Input: n numbers $a_1, a_2, \ldots, a_n$ in array $A[1 \ldots n]$.
Output: n numbers $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$ in array A.

```
1: Sort(A, 1, n)
2: return A[1...n]
```

Procedure Sort(A, p, r).
% Sort $r - p + 1$ numbers $a_p, a_{p+1}, \ldots, a_r$ in array $A[p \ldots r]$. %

```
1: if p < r then
2:     q ← ⌊(p + r)/2⌋
3:     Sort(A, p, q)
4:     Sort(A, q + 1, r)
5:     Merge(A, p, q, r)
6: end if
7: return A[p...r]
```

Procedure Merge(A, p, q, r).
% Merge sorted two arrays $A[p \ldots q]$ and $A[p+1 \ldots r]$ into one. %

```
1: for i ← 1 to q − p + 1 do
2:     B[i] ← A[p + i − 1]
3: end for
4: i ← 1
5: j ← p + 1
6: B[q − p + 2] ← +∞
7: A[r + 1] ← +∞
8: for k ← p to r do
9:     if B[i] ≤ A[j] then
10:        A[k] ← B[i]
11:        i ← i + 1
12:    else
13:        A[k] ← A[j]
14:        j ← j + 1
15:    end if
16: end for
17: return A[p...r]
```

inputs into subproblems with larger inputs. This computation process on input $\{5, 2, 7, 4, 6, 8, 1, 3\}$ is shown in Fig. 2.1.

Note that the running time of procedure Merge at each level is $O(n)$. Let $t(n)$ be the running time of merge sort on input of size n. By the recursive structure, we can obtain that $t(1) = 0$ and

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + O(n).$$

Suppose

$$t(n) \leq 2 \cdot t(\lceil n/2 \rceil) + c \cdot n$$

for some positive constant c. Define $T(1) = 0$ and

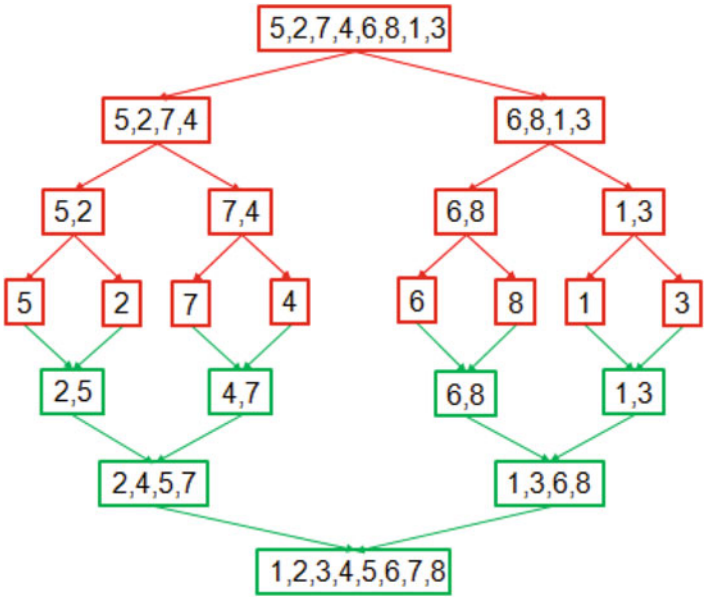


Fig. 2.1 Computation process of merge sort

$$T(n) = 2 \cdot T(\lceil n/2 \rceil) + c \cdot n.$$

By induction, we can prove that

$$t(n) \leq T(n) \text{ for all } n \geq 1.$$

For base step, $t(1) = 0 = T(1)$. For induction step,

$$\begin{aligned} t(n) &\leq 2 \cdot t(\lceil n/2 \rceil) + c \cdot n \\ &\leq 2 \cdot T(\lceil n/2 \rceil) + c \cdot n \quad \text{(by induction hypothesis)} \\ &= T(n). \end{aligned}$$

Now, let us discuss how to solve recursive equation about $T(n)$. Usually, we use two stages. In the first stage, we consider special numbers $n = 2^k$ and employ the recursive tree to find $T(2^k)$ (Fig. 2.2), that is,

$$\begin{aligned} T(2^k) &= 2 \cdot T(2^{k-1}) + c \cdot 2^k \\ &= 2 \cdot (2 \cdot T(2^{k-2}) + c \cdot 2^{k-1}) + c \cdot 2^k \\ &= \cdots \\ &= 2^k T(1) + kc \cdot 2^k \end{aligned}$$

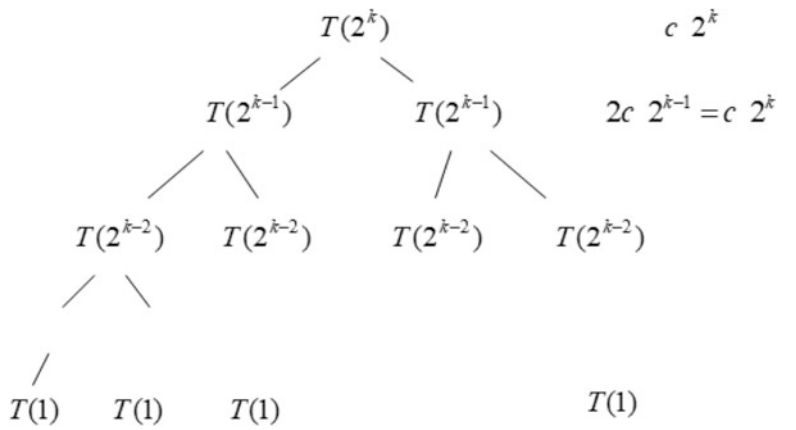


Fig. 2.2 Recursive tree

$$= c \cdot k2^k.$$

In general, we may guess that $T(n) \leq c' \cdot n \log n$ for some constant $c' > 0$. Let us show it by mathematical induction.

First, we choose c' to satisfy $T(n) \leq c'$ for $n \leq n_0$ where n_0 will be determined later. This choice will make $T(n) \leq c'n \log n$ for $n \leq n_0$, which meets the requirement for the basic step of mathematical induction.

For induction step, consider $n \geq n_0 + 1$. Then we have

$$\begin{aligned}
T(n) &= 2 \cdot T(\lceil n/2 \rceil) + c \cdot n \\
&\leq 2 \cdot c' \lceil n/2 \rceil \log \lceil n/2 \rceil + c \cdot n \\
&\leq 2 \cdot c'((n+1)/2)(\log(n+1) - 1) + c \cdot n \\
&= c' \cdot (n+1) \log(n+1) - c'(n+1) + c \cdot n \\
&\leq c'(n+1)(\log n + 1/n) - (c' - c)n - c' \\
&= c'n \log n + c' \log n - (c' - c)n + c'/n.
\end{aligned}$$

Now, we choose n_0 sufficiently large such that $n/2 > \log n + 1/n$ and $c' > \max(2c, T(1), \ldots, T(n_0))$. Then the above mathematical induction proof will be passed. Therefore, we obtained the following.

Theorem 2.1.1 *Merge sort runs in $O(n \log n)$ time.*

By the mathematical induction, we can also prove the following result.

Theorem 2.1.2 *Let $T(n) = aT(n/b) + f(n)$ where constants $a > 1$, $b > 1$, and n/b mean $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$. Then we have the following:*

1. *If* $f(n) = O(n^{\log_b a-\varepsilon})$ *for some positive constant* ε, *then* $T(n) = \Theta(n^{\log_b a})$.
2. *If* $f(n) = \Theta(n^{\log_b a})$, *then* $T(n) = \Theta(n^{\log_b a} \log n)$.
3. *If* $f(n) = \Omega(n^{\log_b a+\varepsilon})$ *for some positive constant* ε *and moreover,* $af(n/b) \leq cf(n)$ *for sufficiently large* n *and some constant* $c < 1$, *then* $T(n) = \Theta(f(n))$.

In Fig. 2.1, we see a tree structure between problem and subproblems. In general, for any algorithm with self-reducibility, its computational process will produce a set of subproblems on which we can also construct a graph to describe relationship between them by adding an edge from subproblem A to subproblem B if at an iteration, subproblem A is reduced to several problems, including subproblem B. This graph is called the *self-reducibility structure* of the algorithm.

All algorithms with tree self-reducibility structure form a class, called *divide-and-conquer*, that is, **an algorithm is in class of divide-and-conquer if and only if its self-reducibility structure is a tree.** Thus, the merge sort is a divide-and-conquer algorithm.

In a divide-and-conquer algorithm, it is not necessary to divide a problem evenly or almost evenly. For example, we consider another sorting algorithm, called *Quick Sort*. The idea is as follows.

In merge sort, the procedure Merge takes $O(n)$ time, which is the main consumption of time. However, if $A[i] \leq A[q]$ for $p \leq i < q$ and $A[q] \leq A[j]$ for $q < j \leq r$, then this procedure can be skipped, and after sort $A[p \ldots q-1]$ and $A[q+1 \ldots r]$, we can simply put them together to obtain sorted $A[p \ldots r]$.

In order to have above property satisfied, Quick Sort uses $A[r]$ to select all elements $A[p \ldots r-1]$ into two subsequences such that one contains elements less than $A[r]$ and the other one contains elements at least $A[r]$. A pseudocode of quick sort is as shown in Algorithm 2.

The division is not balanced in Quick Sort. In the worst case, one part contains nothing, and the other contains $r - p$ elements. This will result in running time $O(n^2)$. However, Quick Sort has expected running time $O(n \log n)$. To see it, let $T(n)$ denote the running time for n numbers. Note that the procedure Partition runs in linear time. Then, we have

$$\begin{aligned} E[T(n)] \leq \frac{1}{n}(& \quad E[T(n-1)] + c_1 n) \\ & + \frac{1}{n}(E[T(1)] + E[T(n-2)] + c_1 n) \\ & + \cdots \\ & + \frac{1}{n}(E[T(n-1)] \quad + c_1 n) \\ = c_1 n + \frac{2}{n} & \sum_{i=1}^{n-1} E[T(i)]. \end{aligned}$$

Algorithm 2 Quick sort

Input: n numbers $a_1, a_2, \ldots, a_n$ in array $A[1 \ldots n]$.
Output: sorted numbers $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$ in array A.

```
Quicksort(A, 1, n)
return A[1...n]
```

Procedure Quicksort(A, p, r).
% Sort $r - p + 1$ numbers $a_p, a_{p+1}, \ldots, a_r$ in array $A[p \ldots r]$. %

```
if p < r then
    q ← Partition(A, p, r)
    Quicksort(A, p, q − 1)
    Quicksort(A, q + 1, r)
end if
return A[p...r]
```

Procedure Partition(A, p, r).
% Find q such that there are $q - p + 1$ elements less than $A[r]$ and others bigger than or equal to $A[r]$ %

```
x ← A[r]
i ← p − 1
for j ← p − 1 to r − 1 do
    if A[j] < x then
        i ← i + 1 and exchange A[i] ↔ A[j]
    end if
    exchange A[i + 1] ↔ A[r]
end for
return i + 1
```

Now, we prove by induction on n that

$$E[T(n)] \leq cn \log n$$

for some constant c. For $n = 1$, it is trivial. Next, consider $n \geq 2$. By induction hypothesis,

$$\begin{aligned}
E[T(n)] &\leq c_1 n + \frac{2c}{n} \sum_{i=1}^{n-1} i \log i \\
&= c_1 n + c(n-1) \log \left(\Pi_{i=1}^{n-1} i^i \right)^{2/(n(n-1))} \\
&\leq c_1 n + c(n-1) \log \frac{1^2 + 2^2 + \cdots + (n-1)^2}{n(n-1)/2} \\
&= c_1 n + c(n-1) \log \frac{2n-1}{3} \\
&\leq c_1 n + cn \log \frac{2n}{3}
\end{aligned}$$

$$= cn \log n + \left(c_1 - c \log \frac{3}{2} \right) n.$$

Choose $c \geq c_1 / \log \frac{3}{2}$. We obtain $E[T(n)] \leq cn \log n$.

Theorem 2.1.3 *Expected running time of Quick Sort is $O(n \log n)$.*

2.2 Rectilinear Minimum Spanning Tree

Consider two points $A = (x_1, y_1)$ and $B = (x_2, y_2)$ in the plane. The *rectilinear distance* of A and B is defined by

$$d(A, B) = |x_1 - x_2| + |y_1 - y_2|.$$

The *rectilinear plane* is the plane with the rectilinear distance, denoted by L_1-plane. In this section, we study the following problem.

Problem 2.2.1 (Rectilinear Minimum Spanning Tree) *Given n points in the rectilinear plane, compute the minimum spanning tree on those n given points.*

In Chap. 1, we already present Kruskal algorithm which can compute a minimum spanning tree within $O(m \log n)$ time. In this section, we will improve this result by showing that the rectilinear minimum spanning tree can be computed in $O(n \log n)$ time. To do so, we first study an interesting problem as follows.

Problem 2.2.2 (All Northeast Nearest Neighbors) *Consider a set P of n points in the rectilinear plane. For each $A = (x_A, y_A) \in P$, another point $B = (x_B, y_B) \in P$ is said to lie in northeast (NE) area of A if $x_A \leq x_B$ and $y_A \leq y_B$, but $A \neq B$. Furthermore, B is the NE nearest neighbor of A if B has the shortest distance from A among all points lying in the NE area of A. This problem is required to compute the NE nearest neighbor for every point in P. (The NE nearest neighbor of a point A is "none" if no given point lies in the northeast area of A.)*

Let us design a divide-and-conquer algorithm to solve this problem. For simplicity of description, assume all n points have distinct x-coordinates and distinct y-coordinates. Now, we bisect n points by a vertical line L. Let P_l be the set of points lying on the left side of L and P_r the set of points lying on the right side of L. Suppose we already solve the all NE nearest neighbors problem on input point sets P_l and P_r, respectively. Let us discuss how to combine solutions for two subproblems into a solution for all NE nearest neighbors on P.

For point A in P_r, the NE nearest neighbor in P_r is also the NE nearest neighbor in P. However, for point A in P_l, the NE nearest neighbor in P_l may not be the NE nearest neighbor in P. Actually, let B_1 denote the NE nearest neighbor of A in P_l and B_r the NE nearest neighbor of A for B_2 in P_r. Then, if $d(A, B_1) \leq d(A, B_2)$,

then the NE nearest neighbor of A in P is B_1; otherwise, it is B_2. Therefore, to complete the combination task, it is sufficient to compute the NE nearest neighbors in P_r for all points in P_l. We will show that this computation takes $O(n)$ time. To do so, let us first show a lemma.

Lemma 2.2.3 *Consider four points A, B, C, and D in the rectilinear plane. Suppose C and D lie in the northeast area of A and the northeast area of B. Then* $d(A, C) \leq d(A, D)$ *if and only if* $d(B, C) \leq d(B, D)$.

Proof Clearly, we have

$$\begin{aligned} & d(A, C) = (x_C - x_A) + (y_C - y_A) \leq d(A, D) = (x_D - x_A) + (y_D - y_A) \\ \Leftrightarrow\ & x_C + y_C \leq x_D + y_D \\ \Leftrightarrow\ & d(B, C) = (x_C - x_B) + (y_C - y_B) \leq d(B, D) = (x_D - x_B) + (y_D - y_B). \end{aligned}$$

□

By this lemma, we can compute the NE nearest neighbors in P_r for all points in P_l as follows.

- For P_l, put all points in decreasing ordering of y-coordinates. For P_r, also, put all points in deceasing ordering of y-coordinates. Put *none* in P_r as the first element. Assume that *none* has y-coordinate $+\infty$ and for any point $A \in P_l$, $d(A, none) = +\infty$.
- Employ three pointers *left*, *right*, and *min*. *left* will be located in P_l. *right* and *min* work in P_r and *none*.
- Initially, assign *left* with the first point in P_l, and assign *right* and *min* with the first element P_r.
- If *right* has y-coordinate higher than *left* and $d(left, right) \geq d(left, min)$, then move *right* to next point in P_r.
- If *right* has y-coordinate higher than *left* and $d(left, right) < d(left, min)$, then set *min* = *right*, and move *right* to next one in P_r.
- If *right* has y-coordinate lower than *left*, then *min* is the NE nearest neighbor of *left*. Put this fact in record, and move *left* to next one in P_l.

Since *left*, *right*, and *min* always move down and never move up, above procedure runs in $O(n)$ time. Let $T(n)$ be the running time for computing all NE nearest neighbors for n points. Then we obtain $T(n) = 2T(\lceil n/2 \rceil) + O(n)$. Therefore, $T(n) = O(n \log n)$.

We make a remark on the case that P contains points with the same x-coordinate or y-coordinates. If P has some points with the same x-coordinate, then in order to partition P into two even parts, we may also consider their y-coordinates. If P has some points with same y-coordinate, then we may need to give a little adjustment for combination procedure.

Theorem 2.2.4 *Computing all NE nearest neighbors for n points can be done in* $O(n \log n)$ *time.*

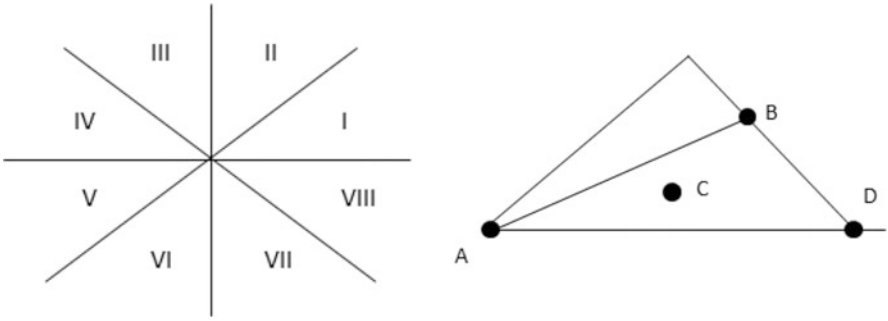


Fig. 2.3 Eight octants of point A

Now, we move back our attention to the rectilinear minimum spanning tree.

Consider any point A. As shown in Fig. 2.3, divide the area surrounding A into eight octants. To make them disjoint, we assume that each octant contains only one boundary, which can be reached from an interior ray to turn in counterclockwise direction.

Lemma 2.2.5 *Suppose (A, B) is an edge in a rectilinear minimum spanning tree. Then B must be the nearest neighbor of A in an octant.*

Proof Without loss of generality, assume that B lies in octant I of point A. For contradiction, suppose that B is not the nearest neighbor of A in octant I. Let C be the nearest neighbor of A in octant I, i.e., C lies in octant I and $d(A, C) < d(A, B)$. Note that

$$\{D \in \text{octant I} \mid d(A, D) \leq d(A, B)\}$$

is a triangle without boundary (A, B), which has a property that every two points in this triangle has rectilinear distance less than $d(A, B)$ (Fig. 2.3). Therefore, $d(C, B) < d(A, B)$.

Remove edge (A, B) from the rectilinear minimum spanning tree, which will partition the tree into two connected components, containing points A and B, respectively. If A and C lie in the same component, then add edge (C, B); otherwise, C and B must lie in the same component, and add edge (A, C). Therefore, we will obtain a shorter rectilinear minimum spanning tree, a contradiction. □

Construct a graph G in the following way: For each point A, if an octant of A contains another given point, then find a nearest neighbor B for A in this octant, and add edge (A, B) to G.

Lemma 2.2.6 *G contains a rectilinear minimum spanning tree.*

Proof Consider a rectilinear minimum spanning tree T. For each point A, T must contain an edge (A, B). By Lemma 2.2.5, B is the nearest neighbor of A in an octant. Suppose (A, B) is not an edge of G. Then G must contain an edge (A, C)

lying in the same octant, and C is another nearest neighbor of A in the same octant. Note that

$$d(A, B) = d(A, C) > d(B, C).$$

Delete (A, B) from tree T. Then T is partitioned into two connected components. We claim that C and B must lie in the same component. In fact, otherwise, assume that A and C lie in the same component. Then we can shorten T by replacing (A, B) with (B, C), contradicting the minimality of T. Therefore, our claim is tree. It follows that replacing (A, B) by (A, C) in T results in another minimum spanning tree T'. Continue above operations; we will find a rectilinear minimum spanning tree contained in G. □

Lemma 2.2.7 *G can be constructed in $O(n \log n)$ time.*

Proof For each point A, its octant represents a direction. Fix an octant, i.e., fix a direction. We claim that computing all octant nearest neighbors for all given points in P takes $O(n \log n)$ time. Without loss of generality, consider octant I. Define a mapping

$$\phi : (x, y) \to ((x - y)/2, y).$$

Then ϕ has the following properties:

- Point B lies in octant I of point A if and only if $\phi(B)$ lies in the first quadrant of $\phi(A)$.
- $d(A, B) \leq d(A, C)$ if and only $d(\phi(A), \phi(B)) \leq d(\phi(A), \phi(C))$.

It follows from those properties that B is the octant nearest neighbor of A if and only if $\phi(B)$ is the NE nearest neighbor of $\phi(A)$. This means that computing all octant nearest neighbors can be reduced to computing all NE nearest neighbors. By Theorem 2.2.4, this computation can be done in $O(n \log n)$ time.

For each of eight octants, $O(n \log n)$ time is required. The total time is still bounded by $O(n \log n)$. □

Theorem 2.2.8 *The rectilinear minimum spanning tree can be computed in $O(n \log n)$ time where n is the number of given points.*

Proof By Lemmas 2.2.6 and 2.2.7, it is sufficient to compute the minimum spanning tree of graph G with Kruskal algorithm, since the number of edges in G is bounded by $O(n)$. Kruskal algorithm will spend $O(n \log n)$ time on G. □

2.3 Fibonacci Search

Consider a sequence of n distinct integers which are stored in an array $A[1..n]$. An element $A[i]$ is a *local maximum* if $A[i-1] < A[i]$ and $A[i] > A[i+1]$ for $1 < i < n$, $A[i] > S[i+1]$ for $i = 1$, and $A[i-1] < A[i]$ for $i = n$. The sequence $A[1..n]$ is said to be *bitonic* if it contains exactly one local maximum, which is actually the global maximum one. Consider the following problem.

Problem 2.3.1 (Maximum Element in Bitonic Sequence) *Given a sequence $A[1..n]$ of n distinct integers, find the maximum element.*

The problem can be solved by the following lemma.

Lemma 2.3.2 *Assume $1 \leq i < j \leq n$. If $A[i] < A[j]$, then $A[i+1..n]$ must contain a local maximum. If $A[i] > A[j]$, then $A[1..j-1]$ must contain a local maximum.*

Proof First, assume $A[i] < A[j]$. Consider two cases.

Case 1. $A[j] < A[j+1]$. In this case, if none of $A[j+1], \ldots, A[n-1]$ is a local maximum, then $A[j+1] < A[j+2] < \cdots < A[n]$. Hence, $A[n]$ is a local maximum.

Case 2. $A[j] > A[j+1]$. In this case, if none of $A[j], A[j-1], \ldots, A[i-1]$ is a local maximum, then $A[j] < A[j-1] < \cdots < A[i]$, contradicting to $A[i] < A[j]$.

Similarly, we can show the second statement. □

For $n \geq 4$, we can choose i and j such that $1 \leq i < j \leq n$, $i \geq n/3$, and $n - j + 1 \geq n/3$. With such i and j, for each comparison, the sequence can be cut off at least one third. Therefore, the maximum element can be found within $O(\log n)$ comparisons.

Next, we consider a situation that $A[i] = f(i)$, that is, $A[i]$ has to be obtained through evaluation of a function $f(i)$. Therefore, we want to find the maximum element with the minimum number of evaluations. In this situation, i and j will be selected based on a rule with Fibonacci number F_i defined as follows:

$$F_0 = F_1 = 1, F_i = F_{i-2} + F_{i-1} \text{ for } i \geq 2.$$

Associated Fibonacci search method is as follows.

Step 0. Select the maximum m such that $F_m \leq n$. Set $k \leftarrow 0$.
Step 1. Set $i \leftarrow k + F_{m-1}$ and $j \leftarrow k + F_m$.
Step 2. If $A[i] < A[j]$, then set $k \leftarrow i$.
Step 3. Set $m \leftarrow m - 1$.
Step 4. If $m \geq 2$, then go back to Step 1. Else, return $A[i+1]$, i.e., $A[k+1]$ is the maximum element.

About this method, we have the following result.

Theorem 2.3.3 *Let m be the maximum integer such that $F_m \leq n$. Then, it is sufficient to make m evaluations to search the maximum element from a bitonic sequence of n distinct integers. Moreover, in the worst case, it is necessary to make m evaluations.*

Proof Sufficiency can be seen from the Fibonacci search algorithm. We prove the necessity by induction on m. For $m = 1$, we have $n = 1$, and evaluation for $A[1]$ is required. For $m = 2$, we have $n = 2$, and evaluations for $A[1]$ and $A[2]$ are necessary. For $m \geq 3$, suppose we compare $A[i]$ and $A[j]$ for $1 \leq i < j \leq n$. Consider two cases.

Case 1. $i \leq F_{m-2}$. In this case, $n - i \geq F_{m-1}$. If $A[i] < A[j]$, then the maximum element is in $A[i+1..n]$. By induction hypothesis, in the worst case, $m-1$ evaluations are required to find the maximum element. Add $A[i]$. Total number of evaluations is m.

Case 2. $i > F_{m-2}$. If $A[i] > A[j]$, then the maximum element is in $A[1..j-1]$. In the next step, we need to select a number $k \in \{1, \ldots, i-1\} \cup \{i+1, \ldots, j-1\}$ and compare $A[i]$ and $A[k]$. It does not matter if $k \in \{1, \ldots, i-1\}$ or $k \in \{i+1, \ldots, j-1\}$, we can have a comparison result to keep $A[1..i-1]$ left. Since $i - 1 \geq F_{m-2}$. In the worst case, we need at least $m-2$ evaluations to find the maximum element from the subsequence containing $A[1..i-1]$. Add evaluations on $A[i]$ and $A[j]$. Total number of required evaluations is at least m. □

2.4 Heap

Heap is a quite useful data structure. Let us introduce it here and, by the way, give another sorting algorithm, Heap Sort.

A heap is a nearly complete binary tree, stored in an array (Fig. 2.4). What is *nearly complete binary tree*? It is a binary tree satisfying the following conditions:

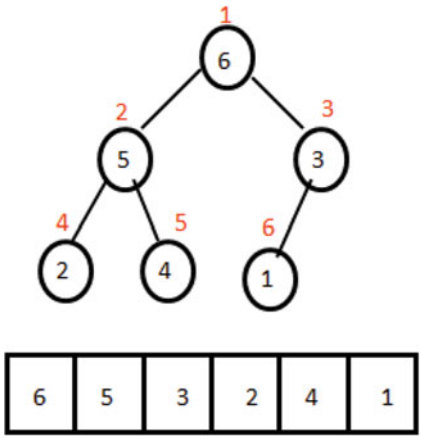
- Every level other than bottom is complete.
- On the bottom, nodes are placed as left as possible.

For example, binary trees in Fig. 2.5 are not nearly complete. An advantage of nearly complete binary tree is to operate on it easily. For example, for node i (i.e., a node with address i), its parent, left-child, and right-child can be easily figured out as follows:

Parent(i)
  return $\lfloor i/2 \rfloor$.

Left(i)
  return $2i$.

Fig. 2.4 A heap



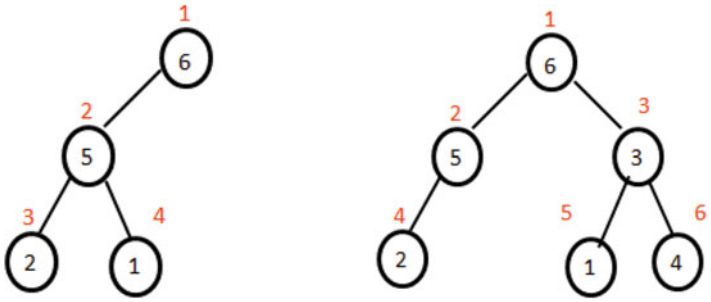


Fig. 2.5 They are not nearly complete

Right(i)
return $2i + 1$.

There are two types of heaps with special properties, respectively.

Max-heap: For every node i other than root, $A[\text{Parent}(i)] \geq A[i]$.
Min-heap: For every node i other than root, $A[\text{Parent}(i)] \leq A[i]$.

Max-heap has two special operations: Max-Heapify and Build-Max-Heap. We describe them as follows.

When operation Max-Heapify(A, i) is called, two subtrees rooted, respectively, at Left(i) and Right(i) are max-heaps, but $A[i]$ may not satisfy the max-heap property. Max-Heapify(A, i) makes the subtree rooted at $A[i]$ become a max-heap by moving $A[i]$ downside. An example is as shown in Fig. 2.6.

The following is an algorithmic description for this operation.

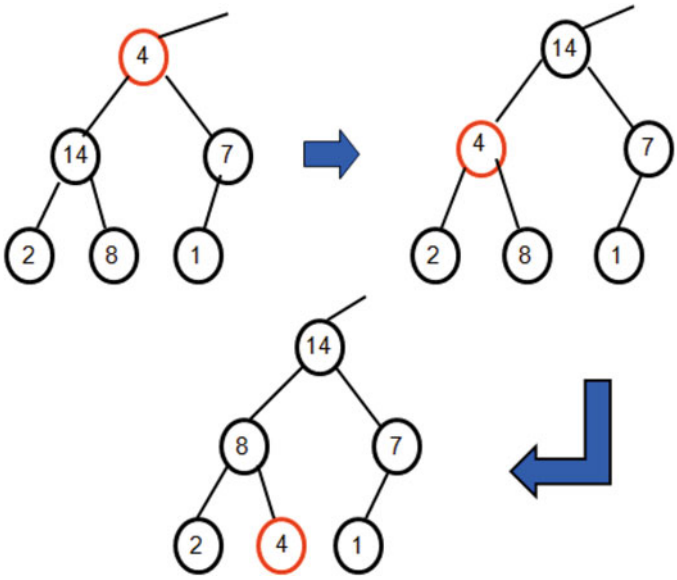


Fig. 2.6 An example for Max-Heapify(A, i)

Max-Heapify(A, i)
if Left(i) ≥Right(i) and Left$(i) > A[i]$
then Exchange $A[i]$ and Left(i)
Max-Heapify$(A,$ Left$(i))$
if Left(i) <Right(i) and Right$(i) > A[i]$
then Exchange $A[i]$ and Right(i)
Max-Heapify$(A,$ Right$(i))$;

Operation Build-Max-Heap applies to a heap and makes it become a max-heap, which can be described as follows. (Note that Parent$(size[A]) = \lfloor size[A]/2 \rfloor$.)

Build-Max-Heap(A)
for $i \leftarrow \lfloor size[A]/2 \rfloor$ down to 1
do Max-Heapify(A, i);

An example is as shown in Fig. 2.7.

It is interesting to estimate the running time of this operation. Let h be the height of heap A. Then $h = \lfloor \log_2 n \rfloor$. At level i, A has 2^i nodes, at each of which Max-Heapify spends at most $h - i$ steps to float down. Therefore, the running time of Build-Max-Heap(A) is

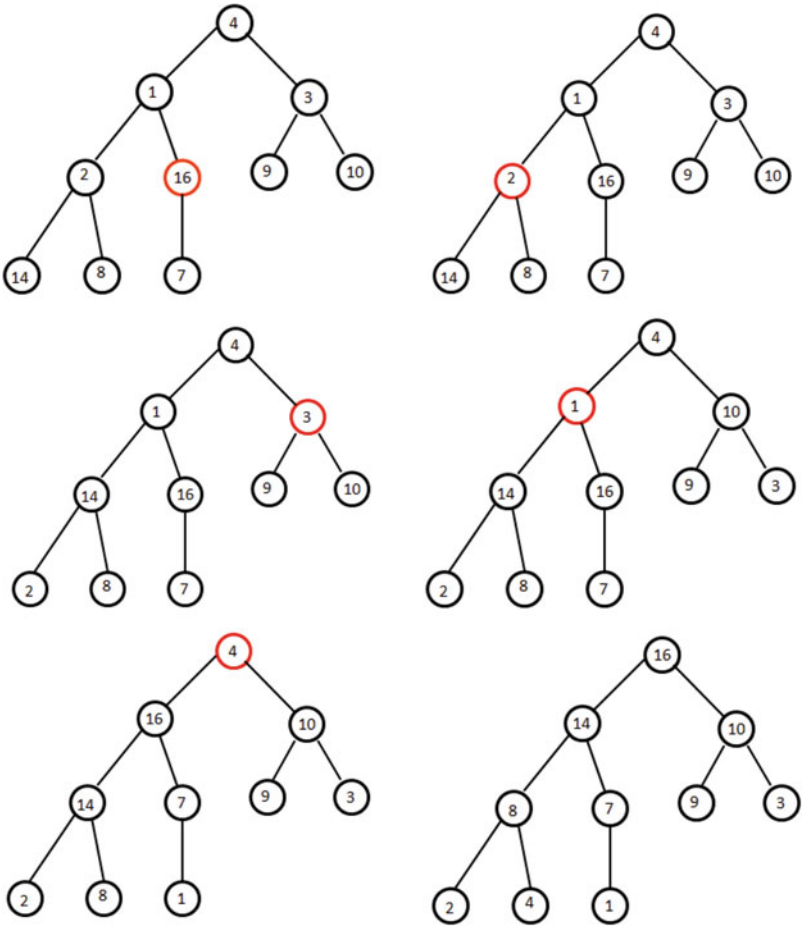


Fig. 2.7 An example for Build-Max-Heap(A)

$$
\begin{aligned}
O\left(\sum_{i=0}^{h} 2^i (h-i)\right) &= O\left(2^h \sum_{i=0}^{h} \frac{h-i}{2^{h-i}}\right) \\
&= O\left(2^h \sum_{i=0}^{h} \frac{i}{2^i}\right) \\
&= O(n).
\end{aligned}
$$

Now, as shown in Algorithm 3, a sorting algorithm can be designed with max-heap. Initially, build a max-heap A. In each subsequent step, the algorithm first exchanges $A[1]$ and $A[heap - size(A)]$ and then reduces $heap - size(A)$ by 1, meanwhile with Max-Heapify$(A, 1)$ to recover the max-heap. An example is as shown in Fig. 2.8.

Algorithm 3 Heap Sort

Input: n numbers $a_1, a_2, \ldots, a_n$ in array $A[1 \ldots n]$.
Output: n numbers $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$ in array A.

1: Build-Max-Heap(A)
2: **for** $i \leftarrow n$ down to 2 **do**
3: exchange $A[1] \leftrightarrow A[i]$
4: heap-size $[A] \leftarrow i - 1$
5: Max-Heapify(A, 1)
6: **end for**
7: **return** $A[1 \ldots n]$

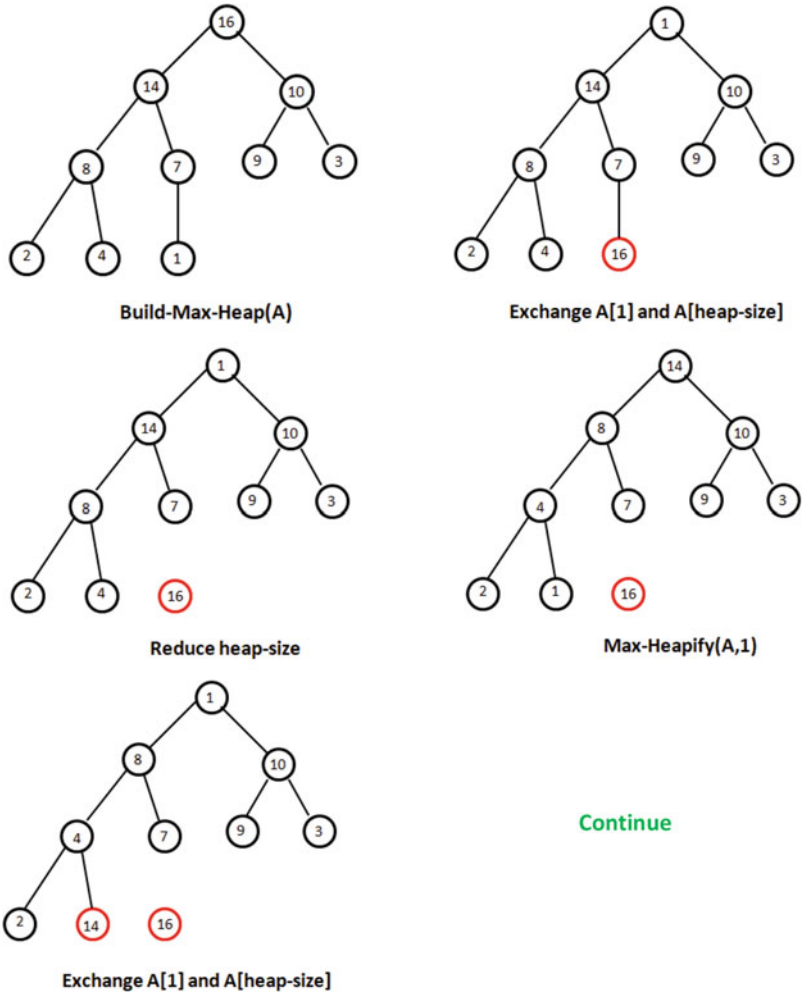


Fig. 2.8 An example for Heap Sort

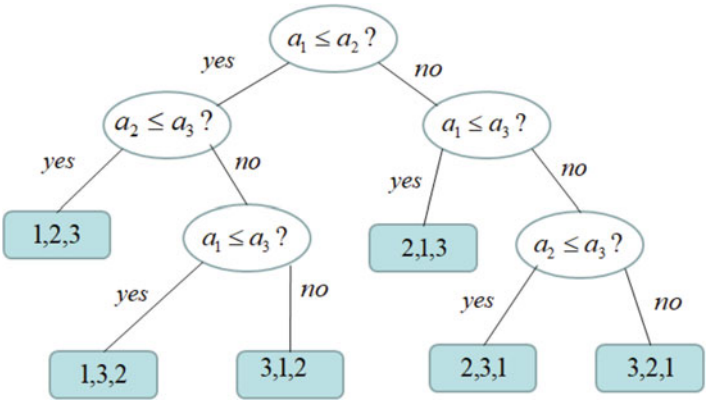


Fig. 2.9 Decision tree

Since the number of steps is $O(n)$ and Max-Heapify$(A, 1)$ takes $O(\log n)$ time, the running time of Heap Sort is $O(n \log n)$.

Theorem 2.4.1 *Heap Sort runs in $O(n \log n)$ time.*

We already have two sorting algorithms with $O(n \log n)$ running time and one sorting algorithm with expected $O(n \log n)$ running time. But, there is no sorting algorithm with running time faster than $O(n \log n)$. Is $O(n \log n)$ a barrier of running time for sorting algorithm? In some sense, the answer is yes. All sorting algorithms presented previously belong to a class, called *comparison sort*.

In comparison sort, order information about input sequence can be obtained only by comparison between elements in the input sequence. Suppose input sequence contains n positive integers. Then there are $n!$ possible permutations. The aim of sorting algorithm is to determine a permutation which gives a nondecreasing order. Each comparison divides the set of possible permutations into two subsets. The comparison result tells which subset contains a nondecreasing order. Therefore, every comparison sort algorithm can be represented by a binary decision tree (Fig. 2.9). The (worst case) running time of the algorithm is the height (or depth) of the decision tree.

Since the binary decision tree has $n!$ leaves, its height $T(n)$ satisfies

$$1 + 2 + \cdots + 2^{T(n)} \ge n!$$

that is,

$$2^{T(n)+1} - 1 \ge \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

Thus,

$$T(n) = \Omega(n \log n).$$

Therefore, no comparison sort can do better than $O(n \log n)$.

Theorem 2.4.2 *The running time of any comparison sort is* $\Omega(n \log n)$.

2.5 Counting Sort

To break the barrier of running time $O(n \log n)$, one has to design a sorting algorithm without using comparison. Counting sort is such an algorithm.

Let us use an example to illustrate Counting Sort as shown in Algorithm 4. This algorithm contains three arrays, A, B, and C. Array A contains input sequence of positive integers. Suppose $A = \{4, 6, 5, 1, 4, 5, 2, 5\}$. Let k be the largest integer in input sequence. Initially, the algorithm makes preprocessing on array C in three stages:

1. Clean up array C.
2. For $1 \leq i \leq k$, assign $C[i]$ with the number of i's appearing in array A. (In the example, $C = \{1, 1, 0, 2, 3, 1\}$ at this stage.)
3. Update $C[i]$ such that $C[i]$ is equal to the number of integers with value at most i appearing in A. (In the example, $C = \{1, 2, 2, 4, 7, 8\}$ at this stage.)

With the help of array C, the algorithm moves element $A[j]$ to array B for $j = n$ down to 1, by

$$B[C[A[j]]] \leftarrow A[j]$$

Algorithm 4 Counting Sort

Input: n numbers $a_1, a_2, \ldots, a_n$ in array $A[1 \ldots n]$.
Output: n numbers $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$ in array B.

```
1: for i ← 1 to k do
2:     C[i] ← 0
3: end for
4: for j ← 1 to n do
5:     C[A[j]] ← C[A[j]] + 1
6: end for
7: for i ← 2 to k do
8:     C[i] ← C[i] + C[i − 1]
9: end for
10: for j ← n down to 1 do
11:     B[C[A[j]]] ← A[j]
12:     C[A[j]] ← C[A[j]] − 1
13: end for
14: return B[1 . . . n]
```

and then updates array C by

$$C[A[j]] \leftarrow C[A[j]] - 1.$$

This part of computation about the example is as follows.

C	1	2	2	4	7	8	
A	4	6	5	1	4	5	$\hat{2}$ $\hat{5}$
B						5	

C	1	2	2	4	6	8	
A	4	6	5	1	4	5	$\hat{2}$ 5
B		2				5	

C	1	1	2	4	6	8	
A	4	6	5	1	4	$\hat{5}$	2 5
B		2			5	5	

C	1	1	2	4	5	8	
A	4	6	5	1	$\hat{4}$	5	2 5
B		2		4		5	5

C	1	1	2	3	5	8	
A	4	6	5	$\hat{1}$	4	5	2 5
B	1	2		4		5	5

C	0	1	2	3	5	8	
A	4	6	$\hat{5}$	1	4	5	2 5
B	1	2		4	5	5	5

C	0	1	2	3	4	8	
A	4	$\hat{6}$	5	1	4	5	2 5
B	1	2		4	5	5	5 6

C	0	1	2	3	4	7	
A	$\hat{4}$	6	5	1	4	5	2 5
B	1	2	4	4	5	5	5 6

Now, let us estimate the running time of Counting Sort.

Theorem 2.5.1 *Counting Sort runs in $O(n + k)$ time.*

Proof The loop at line 1 takes $O(k)$ time. The loop at line 4 takes $O(n)$ time. The loop at line 7 takes $O(k)$ time. The loop at line 10 takes $O(n)$ time. Putting all together, the running time is $O(n+k)$. □

A student found a simple way to improve Counting Sort. Let consider the same example. At the second stage, $C = \{1, 1, 0, 2, 3, 1\}$ where $C[i]$ is equal to the number of i's appearing in array A. The student found that with this array C, array B can be put in integers immediately without array A.

C	1 1 0 2 3 1
B	1
B	1 2
B	1 2 4 4
B	1 2 4 4 5 5 5
B	1 2 4 4 5 5 5 6

Is this method acceptable? The answer is no. Why not? Let us explain.

First, we should note that those numbers in input sequence may come from labels of objects. The same numbers may come from different objects. For example, consider a sequence of objects $\{329, 457, 657, 839, 436, 720, 355\}$. If we use their first digits from left as labels, then we will obtain a sequence $\{9, 7, 7, 9, 6, 0, 5\}$. When apply Counting Sort on this sequence, we will obtain a sequence $\{720, 355, 436, 457, 657, 329, 839\}$. This is because a label gets moved together with its object in Counting Sort.

Moreover, consider two objects 329 and 839 with the same label 9. In input sequence, 329 lies on the left side of 839. After Counting Sort, 329 lies still on the left side of 839.

A sorting algorithm is *stable* if for different objects with the same label, after labels are sorted, the ordering of objects in output sequence is the same as their ordering in input sequence. The following can be proved easily.

Lemma 2.5.2 *Counting Sort is stable.*

The student's method cannot keep stable property.

With stable property, we can use Counting Sort in the following way. Remember, after sorting the leftmost digit, we obtain sequence

$$\{720, 355, 436, 457, 657, 329, 839\}.$$

Now, we continue to sort this sequence based on the second leftmost digit. Then we will obtain sequence

$$\{720, 329, 436, 839, 355, 457, 657\}.$$

Continue to sort based on the rightmost digit, we will obtain sequence

$$\{329, 355, 436, 457, 657, 720, 839\}.$$

Now, let us use this technique to solve a problem.

Example 2.5.3 There are n integers between 0 and $n^2 - 1$. Design an algorithm to sort them. The algorithm is required to run in $O(n)$ time.

Each integer between 0 and $n^2 - 1$ can be represented as

$$an + b \text{ for } 0 \leq a \leq n - 1, 0 \leq b \leq n - 1.$$

Apply Counting Sort first to b and then to a. Each application takes $O(n) = O(n + k)$ time since $k = n$. Therefore, total time is still $O(n)$.

In general, suppose there are n integers, each of which can be represented in the form

$$a_d k^d + a_{d-1} 1 k^{d-1} + \cdots + a_0$$

where $0 \leq a_i \leq k - 1$ for $0 \leq i \leq d$. Then we can sort these n integers by using Counting Sort first on a_0, second on a_1, ..., finally on a_d. This method is called *Radix Sort*.

Theorem 2.5.4 *Radix Sort takes $O(d(n + k))$ time.*

2.6 More Examples

Let us study more examples with divide-and-conquer technique and sorting algorithms.

Example 2.6.1 (Maximum Consecutive Subsequence Sum) Given a sequence of n integers, find a consecutive subsequence with maximum sum.

Divide input sequence S into two subsequence S_1 and S_2 such that $|S_1| = \lfloor n/2 \rfloor$ and $|S_2| = \lceil n/2 \rceil$. Let $MaxSub(S)$ denote the consecutive subsequence of S with maximum sum. Then there are two cases.

Case 1. $MaxSub(S)$ is contained in either S_1 or S_2. In this case, $MaxSub(s) = MaxSub(S_1)$ or $MaxSub(s) = MaxSub(S_2)$.

Case 2. $MaxSub(S) \cap S_1 \neq \emptyset$ and $MaxSub(S) \cap S_2 \neq \emptyset$. In this case, $MaxSub(S) \cap S_1$ is the tail subsequence with maximum sum. That is, suppose $S_1 = \{a_1, a_2, \ldots, a_p\}$. Then among subsequences $\{a_p\}$, $\{a_{p-1}, a_p\}, \ldots, \{a_1, \ldots, a_p\}$, $MaxSub(S) \cap S_1$ is the one with maximum sum. Therefore, it can be found in $O(n)$ time. Similarly, $MaxSub(S) \cap S_2$ is the head subsequence with maximum sum, which can be computed in $O(n)$ time.

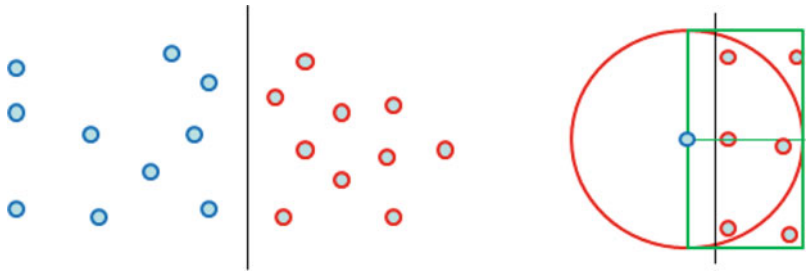


Fig. 2.10 Closest pair of points

Suppose $MaxSub(S)$ can be computed in $T(n)$ time. Summarized from the above two cases, we obtain

$$T(n) = 2T(\lceil n/2 \rceil) + O(n).$$

Therefore, $T(n) = O(n \log n)$.

Next, we present another algorithm running in $O(n)$ time.

Let S_j be the maximum sum of a consecutive subsequence ending at the jth integer a_j. Then, we have

$$S_1 = a_1$$
$$S_{j+1} = \begin{cases} S_j + a_{j+1} & \text{if } S_j > 0, \\ a_{j+1} & \text{if } S_j \leq 0. \end{cases}$$

This recursive formula gives a linear time algorithm to compute S_j for all $1 \leq j \leq n$. From them, find the maximum one, which is the solution for the maximum consecutive subsequence sum problem.

Example 2.6.2 (Closest Pair of Points) Given n points in the Euclidean plane, find a pair of points to minimize the distance between them.

Initially, we may assume that all n points have distinct x-coordinates since, if not, we may rotate the coordinate system a little.

Now, divide all points into two half parts based on x-coordinates. Find the closest pair of points in each part. Suppose δ_1 and δ_2 are distances of closest pairs in two parts, respectively. Let $\delta = \min(\delta_1, \delta_2)$. We next study if there is a pair of points lying in both parts, respectively, and with distance less than δ (Fig. 2.10).

For each point $u = (x_u, y_u)$ in the left part (Fig. 2.10), consider the rectangle $R_u = \{(x, y) \mid x_u \leq x \leq x_u + \delta, y_u - \delta \leq y \leq y_u + \delta\}$. It has the following properties:

- Every point in the right part and within distance δ from u lies in this rectangle.
- This rectangle contains at most six points in the right part because every two points have distance at least δ.

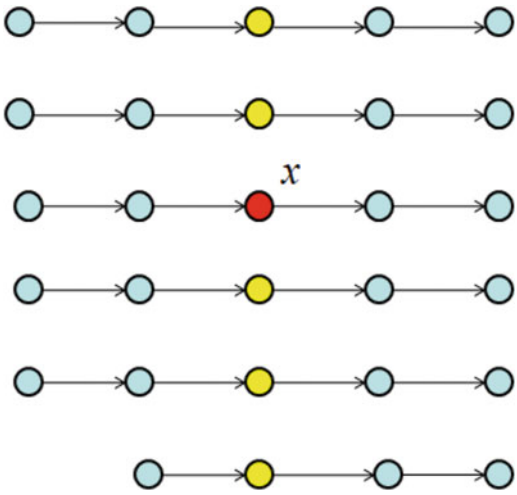


Fig. 2.11 x is selected through first three steps

For each u in the left part, check every point v lying in R_u, if distance $d(u, v) < \delta$. If yes, then we keep the record and choose the closest pair of points from them, which should be the solution. If not, then the solution should either be the closest pair of points in the left part or the closest pair of points in the right part.

Let $T(n)$ be the time for finding the closest pair of points from n points. Above method gives a recursive relation

$$T(n) = 2T(\lceil n/2 \rceil) + O(n).$$

Therefore, $T(n) = O(n \log n)$.

Example 2.6.3 (The ith Smallest Number) Given a sequence of n distinct numbers and a positive integer i, find ith smallest number in $O(n)$ time.

This algorithm consists of five steps. Let us name this algorithm as $A(n, i)$ for convenience of recursive call.

Step 1. Divide n numbers into $\lceil n/5 \rceil$ groups of five elements, possibly except the last one of less than five elements (Fig. 2.11).

Step 2. Find the median of each group by merge sort. Possibly, for the last group, there are two median; in such a case, take the smaller one (Fig. 2.11).

Step 3. Make a recursive call $A(\lceil n/5 \rceil, \lceil \lceil n/5 \rceil /2 \rceil)$. This call will find the median x of $\lceil n/5 \rceil$ group median and, moreover, will select the smaller one in case that two candidates of x exist (Fig. 2.11).

Step 4. Exchange x with the last element in input array, and partition all numbers into two parts by using Partition procedure in Quick Sort. One part (on the left) contains numbers less than x, and the other part (on the right) contains numbers larger than x (Fig. 2.12).

Step 5. Let k be the number of elements in the left part (Fig. 2.12). If $k = i - 1$, then x is the ith smallest number. If $k \geq i$, then the ith smallest number lies on the

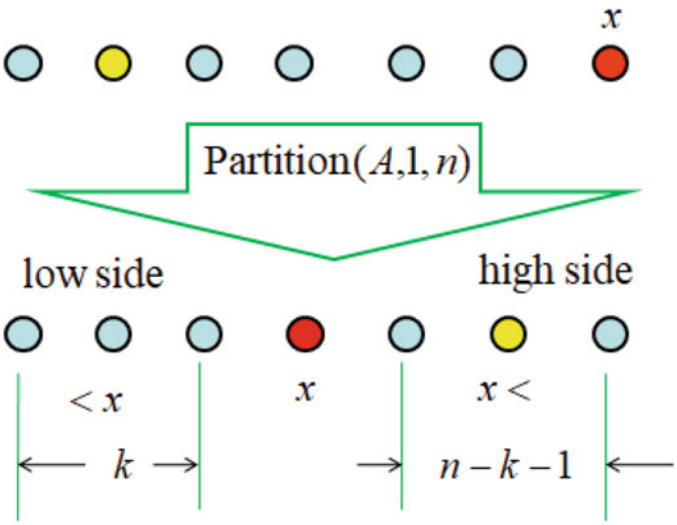


Fig. 2.12 x is selected through the first three steps

left of x and hence makes a recursive call $A(k, i)$. If $k \leq i-2$, then the ith smallest number lies in the right of x and hence makes a recursive call $A(n-k-1, i-k-1)$.

Now, let us analyze this algorithm. Let $T(n)$ be the running time of $A(n, i)$.

- Steps 1 and 2 take $O(n)$ time.
- Step 3 takes $T(\lceil n/5 \rceil)$ time.
- Step 4 takes $O(n)$ time.
- Step 5 takes $T(\max(k, n-k-1))$ time.

Therefore,

$$T(n) = T(\lceil n/5 \rceil) + T(\max(k, n-k-1)) + O(n).$$

We claim that

$$\max(k, n-k-1) \leq n - \left(3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right).$$

In fact, as shown in Fig. 2.13,

$$k+1 = 3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil$$

and

$$n-k \geq 3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2.$$

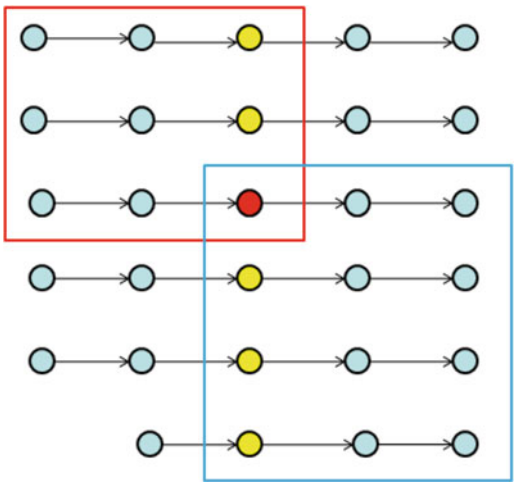


Fig. 2.13 Estimation of $k+1$ and $n-k$

Therefore,

$$n-k-1 \le n-3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil$$

and

$$k \le n-\left(3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil -2\right).$$

Note that

$$n-\left(3\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil -2\right) \le n-\left(\frac{3n}{10}-2\right) \le \frac{7n}{10}+2.$$

By the claim,

$$T(n) \le T(\lceil n/5\rceil)+T\left(\frac{7n}{10}+2\right)+c'n$$

for some constant $c' > 0$. Next, we show that

$$T(n) \le cn \tag{2.1}$$

for some constant $c > 0$. Choose

$$c = \max(20c', T(1), T(2)/2, \ldots, T(59)/59).$$

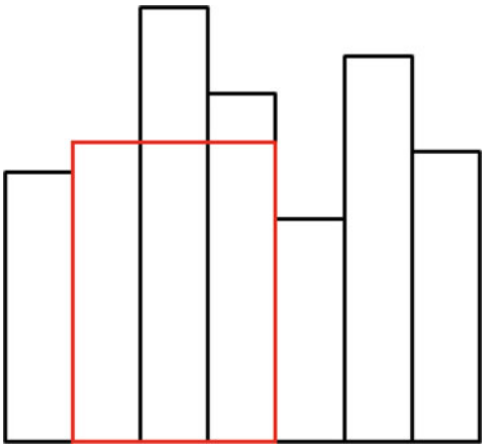


Fig. 2.14 Largest rectangular area in histogram

Therefore, (2.1) holds for $n \leq 59$. Next, consider $n \geq 60$. By induction hypothesis, we have

$$\begin{aligned} T(n) &\leq c(n/5+1) + c(7n/10+2) + c'n \\ &\leq cn - (cn/10 - 3c - c'n) \\ &\leq cn \end{aligned}$$

since

$$c(n/10 - 3) \geq n/20 \geq c'n.$$

The first inequality is due to $n \geq 60$, and the second one is due to $c \geq 20c'$. This ends the proof of $T(n) = O(n)$.

Example 2.6.4 (Largest Rectangular Area in Histogram) Consider a histogram as shown in Fig. 2.14. Assume every bar has unit width and heights are $h_1, h_2, \ldots, h_n$, respectively. Find the largest rectangular area.

Let $h_k = \min(h_i, h_2, \ldots, h_j)$. Denote by $m(i, j)$ the largest rectangular area in histogram with bars between i and j. Then, we can obtain the following recursive formula.

$$m(i, j) = \max(m(i, k-1), m(k+1, j), h_k(j - i + 1)).$$

It is similar to Quicksort that expected running time can be proved to be $O(n \log n)$.

Exercises

1. Use a recursion tree to estimate a good upper bound on the recurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$ and $T(1) = 0$. Use the mathematical induction to prove correctness of your estimation.
2. Draw the recursion tree for $T(n) = 3T(\lfloor n/2 \rfloor) + cn$, where c is a positive constant, and guess an asymptotic upper bound on its solution. Prove your bound by mathematical induction.
3. Show that for input sequence in decreasing order the running time of Quick Sort is $\Theta(n^2)$.
4. Show that Counting Sort is stable.
5. Find an algorithm to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.
6. Let $A[1 : n]$ be an array of n distinct integers sorted in increasing order. (Assume, for simplicity, that n is a power of 2.) Give an $O(\log n)$-time algorithm to decide if there is an integer i, $1 \leq i \leq n$, such that $A[i] = i$.
7. Given an array A of integers, please return an array B such that $B[i] = |\{A[k] \mid k > i \text{ and } A[k] < A[i]\}|$.
8. Given a string S and an integer $k > 0$, find the longest substring of s such that each symbol appears at least k times if it appears in the substring.
9. Given an integer array A, please compute the number of pairs $\{i, j\}$ with $A[i] > 2 \cdot A[j]$.
10. Given a sorted sequence of distinct nonnegative integers, find the smallest missing number.
11. Given two sorted sequences with m, n elements, respectively, design and analyze an efficient divide-and-conquer algorithm to find the kth element in the merge of the two sequences. The best algorithm runs in time $O(\log(\max(m, n)))$.
12. Design a divide-and-conquer algorithm for the following longest ascending subsequence problem: Given an array $A[1..n]$ of natural numbers, find the length of the longest ascending subsequence. (A subsequence is a list $A[i_1]$, $A[i_2], \ldots, A[i_m]$ where m is the length.)
13. Show that in a max-heap of length n, the number of nodes rooted at which the subtree has height h is at most $\lceil \frac{n}{2^{h+1}} \rceil$.
14. Let A be an $n \times n$ matrix of integers such that each row is strictly increasing from left to right and each column is strictly increasing from top to bottom. Give an $O(n)$-time algorithm for finding whether a given number x is an element of A, i.e., whether $x = A(i, j)$ for some i, j.
15. Let S be a set of n points, $p_i = (x_i, y_i)$, $1 \leq i \leq n$, in the plane. A point $p_j \in S$ is a *maximal point of* S if there is no other point $p_k \in S$ such that $x_k \geq x_j$ and $y_k \geq y_j$. In Fig. 2.15, it illustrates the maximal points of a point-set S. Note that the maximal points form a "staircase" which descends rightward. Give an efficient divide-and-conquer algorithm to determine the maximal points of S.
16. Let $A[1..n]$ be an array of n distinct integers where $n \geq 2$. An element $A[i]$ is a local maximum if $A[i - 1] < A[i]$ and $A[i] > A[i + 1]$ for $1 < i < n$,

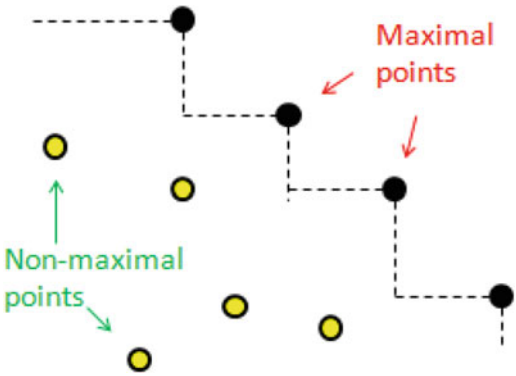


Fig. 2.15 Maximal points and non-maximal points

$A[i] > S[i+1]$ for $i = 1$, and $A[i-1] < A[i]$ for $i = n$. Please design an algorithm to find a local maximum in $O(\log n)$ time.

17. The maximum subsequence sum problem is defined as follows: Given an array $A[1..n]$ of integer numbers, find values of i and j with $1 \le i \le j \le n$ such that $\sum_{k=i}^{j} A[i]$ is maximized. Design a divide-and-conquer algorithm for solving the maximum subsequence sum problem in time $O(n \log n)$.
18. In the plane, there are n distinct points $p_1, p_2, \ldots, p_n$ lying on line $y = 0$ and also n distinct points $q_1, q_2, \ldots, q_n$ lying on line $y = 0$. Consider n segments $[p_1, q_1], [p_2, q_2], \ldots, [p_n, q_n]$. Design an algorithm to count how many cross pairs in these n segments. Your algorithm should run in $O(n \log n)$ time.
19. Design a divide-and-conquer algorithm for multiplying n complex numbers using only $3(n-1)$ real multiplications.
20. Consider a 0-1 matrix of order $(2^n - 1) \times n$. All rows have distinct 0-1 sequences of length n, that is, no two rows are identical. Design a $O(n)$ time algorithm to find the missing sequence.
21. Given a sequence of n distinct integers and a positive integer i, finding the ith smallest one in the sequence can be done in $O(n)$ time (see Example 2.6.3). Now, consider the problem of finding the ith smallest one for every $i = 1, 2, \ldots, k$. Can you do it in $O(n \log k)$ time?
22. An inversion in an array $A[1..n]$ is a pair of indices i and j such that $i < j$ and $A[i] > A[j]$. Design an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time.
23. In Example 2.6.3, a linear time algorithm is given for finding the ith smallest number in a unsorted list of n distinct integers. Now, let us modify the first two steps as follows: Initially, suppose all n integers are given in array A. Partition all input integers into groups of three elements. Then sort each group, and place its median into another array B. Repeat the same process for B, that is, partition elements in B into groups of three elements, and then place the median of each group into array C. Now, make a recursive call to find the median x of C. The remaining part is the same as later steps in the linear time algorithm. Please analyze the running time of this modified algorithm.

24. Design an $O(n^{\log_2 3})$ step algorithms for multiplication of two n-digit numbers. A single step only allows the multiplication/division or addition/subtraction of single digit numbers. Could you improve your algorithm with running $O(n^{\log_3 5})$ steps?

Historical Notes

Divide-and-conquer is a popular technique for algorithm design. It has a special case, decrease-and-conquer. In decrease-and-conquer, the problem is reduced to a single subproblem. Both divide-and-conquer and decrease-and-conquer have a long history. Their stamps can be found in many earlier works, such as Gauss's work on Fourier transform in 1850 [209], John von Neumann's work on merge sort in 1945 [258], and John Mauchly's work in 1946 [258]. Quick Sort was developed by Tony Hoare in 1959 [210] (published in 1962 [211]). Counting Sort and its applications to Radix Sort were found by Harold H. Seward in 1954 [73, 258, 362].

The closest-point problem and its variations, such as the problem of all nearest neighbors, have many applications. Construction of rectilinear minimum spanning tree in $O(n \log n)$ time [192] is one of them. There is another way to obtain $O(n \log n)$-time algorithm for the rectilinear minimum spanning tree [222], in which the Voronoi diagram in L_1 is constructed in $O(n \log n)$ time [274, 363] and then compute the rectilinear minimum spanning tree in the Voronoi diagram in $O(n)$ time. The idea was initiated by Yao [433] to consider closest neighbors in different directions in construction of minimum spanning tree in a plane. In a Euclidean plane, the minimum spanning tree can also be computed in $O(n \log n)$ time [273, 274, 363]. For planar graphs, the minimum spanning tree can be computed in $O(n)$ time [63]. Several algorithms exist for a long time for computing the minimum spanning tree with arbitrary distance [36, 65, 266, 339].

Fibonacci search [140] is motivated from Golden section search [244] to find the maximum value of a unimodal function since $F_k/F_{k+1} \leftarrow (\sqrt{5}-1)/2$, which is called the Golden ratio. The Golden section search has received a great deal of applications [140, 218, 258, 333].