Algorithms

Selected Lecture Notes

Partha Bhowmick

IIT Kharagpur

http://www.facweb.iitkgp.ernet.in/~pb

Foreword

Some selected topics and related algorithms I have discussed in the class are explained briefly in this Lecture Notes. There are also some other topics I have discussed in the class but not included in this Lecture Notes. You should see your class notes and referred books for those. Some typographical errors and notational discrepancies may be there in this Lecture Notes. If you find some, please email me at bhowmick@gmail.com.

pb

Contents

Chapter 1

Search Trees

The figure aside shows a sculpture by Andrew Rogers in Jerusalem. The sculpture ratio is proportioned according to ***Fibonacci numbers***. Interestingly, it has a subtle connection with the ***golden ratio***, which is again connected with search trees (Sec. 1.2).

1.1 Binary Search Tree (BST)

Definition 1.1 (Binary Tree) *A binary tree is a data structure in the form of a* **rooted tree** *in which each node has at most two children. A recursive definition: A binary tree is either empty or it contains a root node together with two binary trees called the left subtree and the right subtree of the root.*

Definition 1.2 (BST) *A BST is either empty or a binary tree with the following properties:*

(i) All keys (if any) in the left subtree of the root precede the key in the root.

(ii) The key in the root precedes all keys (if any) in the right subtree of the root.

(iii) The left and the right subtrees of the root are BST.

C declaration of a binary tree or a BST is as follows.

```
typedef struct tnode {
    int x; //info
    struct tnode *left, *right;
} node;
```

1.1.1 Traversal in a Binary Tree

Let u be any node and L and R be its respective left and right subtrees. Then the following three types of traversals are defined on a binary tree or a BST:

Inorder traversal: Represented as LuR, the left subtree L is (recursively) visited (and nodes reported/processed) first, then the node u, and finally the right subtree R.

Preorder traversal: uLR.

Postorder traversal: LRu.

Note that, by inorder traversal of a BST, we always get the sorted sequence of its keys. C-snippet for inorder traversal is given below. For other two types, it's obvious.

```
void in_order(node *u)
{
    if (u!=NULL){
        in_order(u->left);
        visit(u); //ex: print(u)
        in_order(u->right);
    }
}
```

1.1.2 Searching in a BST

C-snippet for searching is as follows.

```
main()
{
    node *root;
    int k;
    ...
    p = bst_search(root, k);
    ...
}

node *bst_search(node *p, int k)
{
    if (p!=NULL)
        if (k < p->x)
            p = bst_search(p->left, k);
        else if (k > p->x)
            p = bst_search(p->right, k);

    return p;
}
```

The pseudo-code to search a key k in a BST is as follows:

Algorithm SEARCH-BST(r, k)
1. **if** $k < key[r]$
2. SEARCH-BST$(left[r], k)$

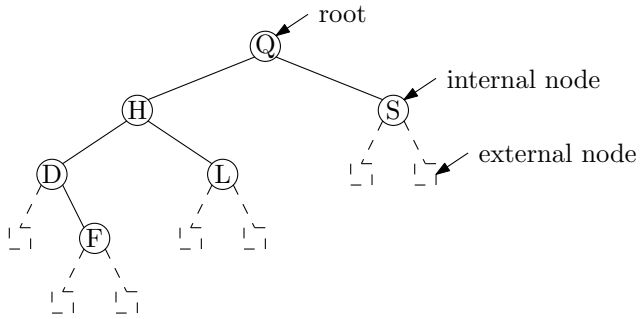


Figure 1.1: An extended BST with $n = 6$ internal nodes and $n + 1 = 7$ external/dummy nodes. It has $I(n) = 1 \times 2 + 2 \times 2 + 3 \times 1 = 9$ and $E(n) = 2 \times 2 + 3 \times 3 + 4 \times 2 = 21$. Hence, for this particular BST, we get $S(n) = (9 + 6)/6 = 2.50$ and $U(n) = 21/7 = 3.00$.

3. **else if** $k > key[r]$
4. SEARCH-BST$(right[r], k)$
5. **return** r

Time complexity

Best case:

$T(n) = O(n)$, since the search key k may be the very root key.

Worst case:

$T(n) = T(n-1) + O(1) = O(n)$, which arises when the BST is fully skewed and the search terminates at the bottommost leaf node.

Average case:

We define the following terms for a BST having n nodes (see Fig. 1.1):
Internal path length: $I(n)$ = Sum of path lengths of all internal nodes from the root (of BST).
External path length: $E(n)$ = Sum of path lengths of all external (dummy) nodes from the root.
Average number of comparisons for successful search: $S(n)$.
Average number of comparisons for unsuccessful search: $U(n)$.

Observe that

$$S(n) = \frac{I(n) + n}{n}, \quad U(n) = \frac{E(n)}{n+1}. \tag{1.1}$$

It can be proved by induction that

$$E(n) = I(n) + 2n. \tag{1.2}$$

Hence,

$$U(n) = \frac{I(n) + 2n}{n+1}. \tag{1.3}$$

Eliminating $I(n)$ from Eqns. 1.1 and 1.3,

$$\begin{aligned} nS(n) &= (n+1)U(n) - n, \\ \text{or,} \quad S(n) &= \left(1 + \frac{1}{n}\right) U(n) - 1. \end{aligned} \tag{1.4}$$

To find the *average* number of comparisons for successfully searching *each* of the n keys, we consider its *insertion order*. If a key x was inserted as the ith node, namely u_i, then the average number of comparisons for its unsuccessful search in that instance of the tree containing the preceding $(i-1)$ nodes, is given by U_{i-1}. Once it's inserted, we can successfully search for it and the search terminates at the node u_i, which was a dummy node where its unsuccessful search terminated. Hence, $S_i = U_{i-1} + 1$, as one extra comparison is required for the successful search terminating at u_i compared to the unsuccessful search terminating at the dummy node located at the same position. We estimate the average number of comparisons for all these n nodes based on their insertion orders. Thus, we get

$$S(n) = \frac{1}{n}\sum_{i=1}^{n}(U_{i-1} + 1), \tag{1.5}$$

From Eqns. 1.4 and 1.5,

$$(n+1)U(n) = 2n + U(0) + U(1) + \ldots + U(n-1).$$

To solve the above recurrence, we substitute $n-1$ for n to get

$$nU(n-1) = 2(n-1) + U(0) + U(1) + \ldots + U(n-2).$$

Subtracting the latter from the former,

$$U(n) = U(n-1) + \frac{2}{n+1}. \tag{1.6}$$

Since $U(0) = 0$, we get $U(1) = \frac{2}{2}$, $U(2) = \frac{2}{2} + \frac{2}{3}$, $U(3) = \frac{2}{2} + \frac{2}{3} + \frac{2}{4}$, and so on, resulting to

$$U(n) = 2\left(\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n+1}\right) = 2H_{n+1} - 2 \approx 2\ln n = (2\ln 2)\log_2 n, \tag{1.7}$$

where, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$ is the nth *harmonic number*, and approximately equals $\ln n$.

Hence, from Eqn. 1.4, we can find $S(n)$, and conclude that $S(n) \approx U(n) \approx (2\ln 2)\log_2 n$.

1.1.3 Insertion in a BST

For insertion of a new element, we search for that element in the BST. It's an unsuccessful search, terminating at a dummy node. A new node is created at the position of that dummy node with necessary pointer adjustments. The C-snippet is given below.

```
main()
{
    node *root, *u;
    int k;
    ...
    if ((u = (node *) malloc(sizeof(node))) == NULL)
        error("Exhausted memory.");
    else {
        u->x = k;
        u->left = u->right = NULL;
    }
```

```
    u = bst_insert(root, u);
    ...
}

node *insert(node *r, node *u)
{
    if (r == NULL)
        r = u;
    else if (u->x < r->x)
        r->left = bst_insert(r->left, u);
    else if (u->x < r->x)
        r->right = bst_insert(r->right, u);
    else
        error("key already exists in BST.");
    return r;
}
```

Time complexity

The best-, worst-, and average-case time complexities for insertion of a node in a BST are all similar as those for (unsuccessful) searching.

1.1.4 Deletion from a BST

While deleting an element k from a BST, we first search for k in the BST, and then delete the corresponding node, say u, based on the following possible cases (see Fig. 1.2).

Case 1: If u is a leaf node, then we simply free u and reassign its parent's child pointer (left or right, as applicable) to NULL.

Case 2: If u has only one subtree, say, the left subtree rooted at v, then we reassign the pointer from its parent p to v (and free u).

Case 3: If u has both the subtrees, the left (T_2) rooted at v and the right (T_3) at w, then we traverse T_3 until we reach its leftmost node, say u', which has no left child. After deletion of u, the node v should be the inorder predecessor of u'. Hence, the left-child pointer from u' is set to v and the pointer from p to u is reset to w (and u is freed).

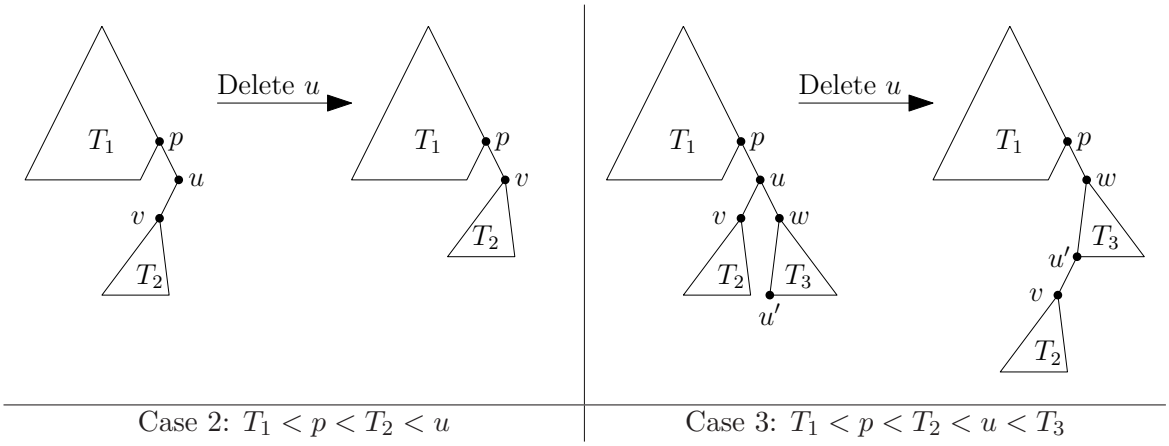
Time complexity

Pointer adjustments in all three cases need constant time. So, the best-, worst-, and average-case time complexities for deletion of a node from a BST are all similar as those for (successful) searching.

1.2 AVL Tree

Definition 1.3 (AVL tree) *An AVL tree*[1] *is a* ***self-balancing binary search tree (BST)*** *in which the heights of the two child subtrees of any node differ by at most one.*

[1]Named after its two inventors, G.M. Adelson-Velskii and E.M. Landis (1962): *An algorithm for the organization of information*, Proceedings of the USSR Academy of Sciences **146**: 263–266 (Russian). English translation by Myron J. Ricci in Soviet Math. Doklady, **3**: 1259–1263, 1962.



Case 2: $T_1 < p < T_2 < u$ Case 3: $T_1 < p < T_2 < u < T_3$

Figure 1.2: Deletion from a BST: Case 2 and Case 3.

In an AVL tree having n nodes, searching, insertion, and deletion all take $O(\log n)$ time in both the average and the worst cases. Insertions and deletions may require the tree to be re-balanced by one or more tree rotations. The ***balance factor*** of a node is the height of its left subtree minus the height of its right subtree (a convention, although the opposite is equally valid). A node with the balance factor 0 or ± 1 is said to be ***balanced***. A node with any other balance factor is considered unbalanced and requires re-balancing. The balance factor is either stored directly at each node or computed from the heights of the subtrees (how!?). In short, a BST is an AVL tree iff each of its nodes is balanced.

AVL trees are often compared with red-black trees because they support the same set of operations and because red-black trees also take $O(\log n)$ time for the basic operations. AVL trees perform better than red-black trees for lookup-intensive applications.[1]

1.2.1 Worst-case AVL (Fibonacci) trees

We find the minimum number of nodes, say n_h, in an AVL tree of height h. Let h be the height of an AVL tree. Then, for the minimality of its node count, we have

$$n_h = n_{h-1} + n_{h-2} + 1, \tag{1.8}$$

where $n_0 = 1$ and $n_1 = 2$, since the minimality of node count demands a similar (recursive) minimality of the subtrees.

By adding 1 to both sides of the above equation, we get the Fibonacci relation

$$F_k = F_{k-1} + F_{k-2}, \quad \text{where } F_0 = 0 \text{ and } F_1 = 1, \tag{1.9}$$

so that $F_k = n_h + 1$, whereby $F_3(= 2)$ in starting correspondence with $n_0 + 1(= 2)$.

To solve the above recurrence, we use the following ***generating function*** with the Fibonacci numbers as coefficients:

$$\begin{aligned} F(x) &= F_0 + F_1 x + F_2 x^2 + \ldots + F_n x^n + \ldots \\ xF(x) &= \quad F_0 x + F_1 x^2 + \ldots + F_{n-1} x^n + \ldots \\ x^2 F(x) &= \quad\quad F_0 x^2 + \ldots + F_{n-2} x^n + \ldots \end{aligned}$$

[1] Ben Pfaff (June 2004). *Performance Analysis of BSTs in System Software*, Stanford University. See `http://www.stanford.edu/~blp/papers/libavl.pdf`.

or, $(1 - x - x^2)F(x) = F_0 + (F_1 - F_0)x = x$,
or,

$$F(x) = \frac{x}{1 - x - x^2} = \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi x} - \frac{1}{1 - \psi x} \right), \tag{1.10}$$

where $\phi = \frac{1}{2}(1 + \sqrt{5})$ and $\psi = \frac{1}{2}(1 - \sqrt{5})$ are the roots of $1 - x - x^2 = 0$. Hence,

$$F(x) = \frac{1}{\sqrt{5}} \left(1 + \phi x + \phi^2 x^2 + \ldots - 1 - \psi x - \psi^2 x^2 - \ldots\right). \tag{1.11}$$

Observe that from the generating function $F(x)$, the coefficient of x_n is F_n, which implies

$$F_n = \frac{1}{\sqrt{5}} \left(\phi^n - \psi^n\right) \approx \frac{1}{\sqrt{5}} \left(1.618^n - (-0.618)^n\right) = int \left(\frac{\phi^n}{\sqrt{5}} \right). \tag{1.12}$$

Note: The term $\phi \approx 1.6180339887498948482...$ is called the ***golden ratio***—a figure where mathematics and arts concur at! Some of the greatest mathematical minds of all ages, from Pythagoras and Euclid in ancient Greece, through the medieval Italian mathematician Leonardo of Pisa and the Renaissance astronomer Johannes Kepler, to present-day scientific figures such as Oxford physicist Roger Penrose, have spent endless hours over this simple ratio and its properties. But the fascination with the Golden Ratio is not confined just to mathematicians. Biologists, artists, musicians, historians, architects, psychologists, and even mystics have pondered and debated the basis of its ubiquity and appeal. In fact, it is probably fair to say that the Golden Ratio has inspired thinkers of all disciplines like no other number in the history of mathematics.

As a continued fraction: $\phi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \ddots}}$

As a continued square root, or infinite surd: $\phi = \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1 + \ldots}}}}$

In simpler terms, two quantities are in the golden ratio if the ratio of the sum of the quantities to the larger quantity is equal to the ratio of the larger quantity to the smaller one; that is, $\phi = \frac{a}{b} = \frac{a+b}{a}$.

Synonyms used for the golden ratio are the golden section (Latin: sectio aurea), golden mean, extreme and mean ratio (by Euclid), medial section, divine proportion (Leonardo da Vinci's illustrations), divine section, golden proportion, golden cut, golden number, and mean of Phidias[1].

One of the many interesting facts about golden ratio: A pyramid in which the apothem (slant height along the bisector of a face) is equal to ϕ times the semi-base (half the base width) is called a *golden pyramid*. Some Egyptian pyramids are very close in proportion to such mathematical pyramids.

1.2.2 Insertion and Deletion from an AVL Tree

Insertion of a node u in an AVL tree consists of two stages: (i) insertion of u as in the case of a BST; (ii) applying single (L or R) or double (LR or RL) rotation at a node where the balance factor is violated (i.e., has become $+2$ or -2 after inserting u). Rotations are explained in Fig. 1.3

Interestingly, it can be shown that—by considering case-wise possibilities—at most one (single or double) rotation will ever be done. If so, then it's near the newly inserted node u.

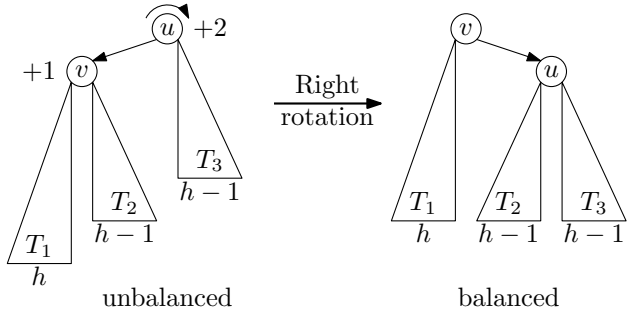
As the height of an AVL tree is $O(\log n)$, insertion takes $O(\log n)$ time for stage (i) and $O(1)$ time for stage (ii), making it $O(\log n)$ in total.

[1]For interesting facts and figures, see `http://en.wikipedia.org/wiki/Golden_ratio`

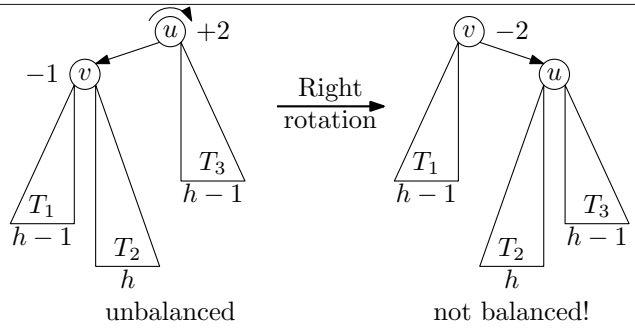
Deletion also consists of two stages as in the case of insertion. The first stage is again similar to that of BST. For the second stage, we have to traverse up to the very root of the BST, go on updating the balance factors of the predecessors along the path to the root, and apply rotations as and when required. The total time complexity for the two stages is, therefore, $O(\log n) + O(\log n) = O(\log n)$.

Suggested Book

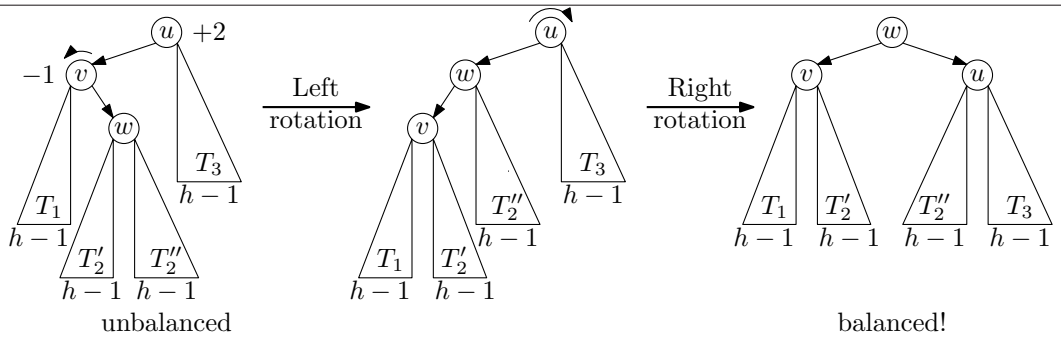
R.L. Kruse, B.P. Leung, C.L. Tondo. *Data Structures and Program Design in C.* PHI, 2000.



Single right (R) rotation: Applied when the node u has balance factor $f = +2$ and its left child v has $f = +1$. Note that the BST property $T_1 \leq v \leq T_2 \leq u \leq T_3$ is preserved after rotation.



A case where single right (R) rotation fails: When the node u has balance factor $f = +2$ and its left child v has $f = -1$, although the BST property $T_1 \leq v \leq T_2 \leq u \leq T_3$ is preserved after rotation. Double right (LR) rotation balances it correctly.



Double right (LR) rotation: Applied when the node u has balance factor $f = +2$ and its left child v has $f = -1$. Note that the BST property $T_1 \leq v \leq T_2' \leq w \leq T_2'' \leq u \leq T_3$ is preserved after rotation.

Figure 1.3: Single right (R) and double right (LR) rotations for balancing an AVL tree. The other two types of rotations, namely single left (L) and double left (RL) rotations, are similar (to be precise, just vertical reflections of R and LR).

Chapter 2

Sorting

"Just as the young squirrel must learn what is climbable and foraging during the dependency period may provide that experience. Beside the selection of food, other feeding behavior such as the (sorting and) opening of nuts, improves with time."—J.P.Hailman

2.1 Insertion Sort

Principle: Iteratively sorts the first i ($2 \leq i \leq n$) elements in a list L using the results of the already-sorted first $i-1$ elements in the previous iteration. To do so, it simply inserts the ith element properly among the previously sorted $i-1$ elements.

Algorithm INSERTION-SORT(L, n)
1. **for** $i \leftarrow 2$ to n
2. $\quad x \leftarrow L[i]$
3. $\quad j \leftarrow i-1$
4. $\quad$ **while** $j > 0 \wedge L[j] > x$
5. $\quad\quad L[j+1] \leftarrow L[j]$
6. $\quad\quad j \leftarrow j-1$
7. $\quad L[j+1] \leftarrow x$

2.1.1 Time complexity

Best case: Minimum number of comparisons to insert the ith ($i > 1$) element $= 1$. Hence, $T(n) = (n-1) \cdot O(1) = O(n)$.

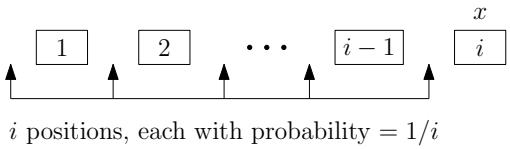
Worst case: Maximum number of comparisons to insert the ith ($i > 1$) element $= i-1$. Hence,

$$T(n) = \sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2} = O(n^2).$$

Average case: The ith element x is equally likely to be placed at any one of the i positions. So, each position has probability $= 1/i$.

Thus, average number of comparisons to insert x is

$$\sum_{j=1}^{i-1}\left(\frac{1}{i}\cdot j\right)+\frac{1}{i}(i-1)=\frac{i+1}{2}-\frac{1}{i}.$$



Note: If x is inserted before or after the very first element, then we need $i-1$ comparisons (see figure aside).

Considering all n passes,

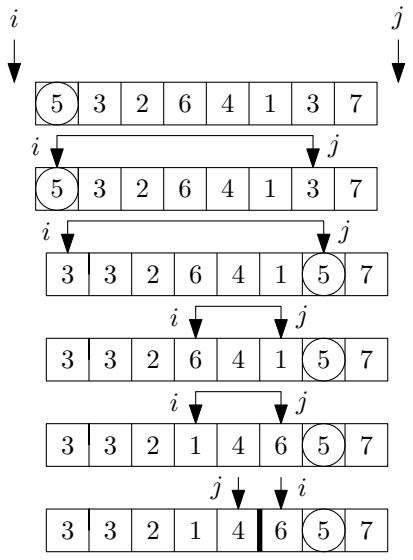
$$\begin{aligned}T(n)=\sum_{i=2}^{n}\left(\tfrac{1}{2}(i+1)-\tfrac{1}{i}\right) &= \tfrac{n^2}{4}+\tfrac{3n}{4}-1-\sum_{2}^{n}\tfrac{1}{i}\\ &= O(n^2)+O(n)-O(\log n)\ \left(\text{since }\sum_{2}^{n}\tfrac{1}{i}=O(\ln n)\right)\\ &= O(n^2).\end{aligned}$$

2.2 Quicksort

Principle: Based on *divide and conquer*, in-place[1] but not stable[2].
Divide The input list $L[1..n]$ is partitioned into two nonempty sublists, $L_1 := L[1..q]$ and $L_2 := L[q+1..n]$, such that no element of L_1 exceeds any element of L_2. The index q is returned from the PARTITION procedure.
Conquer and combine L_1 and L_2 are recursively quicksorted in place so that their final combination is sorted.[3][4]

Algorithm QUICKSORT(L, p, r)
1. **if** $p < r$
2. $\quad q \leftarrow$ PARTITION(L, p, r)
3. $\quad$ QUICKSORT(L, p, q)
4. $\quad$ QUICKSORT$(L, q+1, r)$

Procedure PARTITION(L, p, r)
1. $x \leftarrow L[p]$
2. $i \leftarrow p-1$
3. $j \leftarrow r+1$
4. **while** TRUE
5. $\quad$ **do** $j \leftarrow j-1$
6. $\quad\quad$ **till** $L[j] > x$
7. $\quad$ **do** $i \leftarrow i+1$
8. $\quad\quad$ **till** $L[i] < x$
9. $\quad$ **if** $i < j$
10. $\quad\quad$ SWAP$(L[i], L[j])$
11. $\quad$ **else return** j



[1] An in-place sorting needs no extra array.
[2] A stable sorting preserves the relative order of records with equal keys.
[3] "An Interview with C.A.R. Hoare". Communications of the ACM, March 2009 (premium content).
[4] R. Sedgewick, Implementing quicksort programs, Comm. ACM, 21(10):847–857, 1978.

The correctness of the partition algorithm is based on the following facts: (i) $|L_1| > 0$ and $|L_2| > 0$; (ii) $L_1 \cup L_2 = L$; (iii) $L_1 \leq x \leq L_2$;

The correctness of the sorting algorithm follows from inductive reasoning. For one element, the algorithm leaves the data unchanged; otherwise it produces the concatenation of L_1 and L_2, which are themselves recursively sorted by the inductive hypothesis.

2.2.1 Time complexity

Best case: Partition takes $\Theta(n)$ time. So, best-case time complexity of quicksort is $T(n) = 2T(n/2) + \Theta(n)$, which solves to $T(n) = \Theta(n \log n)$.

Worst case: Arises when $\max(|L_1|, |L_2|) = n - 1$ in every step of the recursion, giving $T(n) = T(n-1) + \Theta(n)$, or, $T(n) = \Theta(n^2)$.

Average case: Based on the assumption that the pivot x is equally likely to be the ith min for $i = 1, 2, \ldots, n$. If x is the 1st min, then x is the sole element in L_1 so that $|L_1| > 0$ (invariant of PARTITION); for all other cases, $x \in L_2$. So, $|L_1| = 1$ occurs when x is the 1st or the 2nd min. Thus, $Prob(|L_1| = 1) = 2/n$, and $Prob(|L_1| = q) = 1/n$ for $q = 2, \ldots, n-1$. Hence, the worst-case time complexity of quicksort is

$$
\begin{aligned}
T(n) &= \frac{1}{n}\left((T(1) + T(n-1)) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n) && (2.1)\\
&\leq \frac{1}{n}\left(O(n^2) + \sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n)\\
&= \frac{1}{n}\left(\sum_{q=1}^{n-1}(T(q) + T(n-q))\right) + \Theta(n), \quad \text{as } \tfrac{1}{n}O(n^2) = O(n) \text{ and } O(n) + \Theta(n) = \Theta(n)\\
&\leq \frac{2}{n}\sum_{q=1}^{n-1} T(q) + \Theta(n) && (2.2)
\end{aligned}
$$

We solve the above equation using the method of substitution (induction), with the hypothesis that $T(q) \leq aq \log q$ for a suitable constant $a > 0$ and $\forall\ q < n$. Then,

$$
\begin{aligned}
T(n) &\leq \frac{2}{n}\sum_{q=1}^{n-1} aq \log q + \Theta(n) \quad = \frac{2a}{n}\sum_{q=1}^{n-1}(q \log q) + \Theta(n)\\
&= \frac{2a}{n}\left(\sum_{q=1}^{\lceil n/2\rceil - 1}(q \log q) + \sum_{q=\lceil n/2\rceil}^{n-1}(q \log q)\right) + \Theta(n)\\
&\leq \frac{2a}{n}\left(\log(n/2)\sum_{q=1}^{\lceil n/2\rceil - 1} q \;+\; \log n \sum_{q=\lceil n/2\rceil}^{n-1} q\right) + \Theta(n)\\
&= \frac{2a}{n}\left((\log n - 1)\sum_{q=1}^{\lceil n/2\rceil - 1} q \;+\; \log n \sum_{q=\lceil n/2\rceil}^{n-1} q\right) + \Theta(n)\\
&= \frac{2a}{n}\left(\log n \sum_{q=1}^{n-1} q \;-\; \sum_{q=1}^{\lceil n/2\rceil - 1} q\right) + \Theta(n)
\end{aligned}
$$

$$
\begin{aligned}
or, \quad T(n) &\leq \frac{2a}{n}\left(\frac{1}{2}n(n-1)\log n \;-\; \frac{1}{2}\left(\left\lceil\frac{n}{2}\right\rceil - 1\right)\left\lceil\frac{n}{2}\right\rceil\right) + \Theta(n) \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n(n-1)\log n \;-\; \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2}\right) + \Theta(n) \\
&= a(n-1)\log n - \frac{a}{2}\left(\frac{n}{2} - 1\right) + \Theta(n) \\
&= an\log n - \left(\frac{a}{2}\left(\frac{n}{2} - 1\right) + a\log n - \Theta(n)\right),
\end{aligned}
$$

which can be made smaller than $an\log n$ for a sufficiently large value of a, $\forall\ n \geq n_0$. Thus, $T(n) = O(n\log n)$.

2.3 Heapsort

A (binary max) ***heap*** is a ***complete binary tree***, i.e., its levels are completely filled except possibly the lowest, which is filled from left to right (*shape property of heap*). Further, (the key at) each of its nodes is greater than or equal to each of its children (*heap property*). It is represented as 1-D array, A, for a simple-yet-efficient implementation as follows (see Fig. 2.1):

Root of the heap = $A[1]$; for each node corresponding to $A[i]$, the parent is $A[\lfloor i/2 \rfloor]$, left child is $A[2i]$, and right child is $A[2i+1]$. Thus, for a heap having n nodes, $A[\lfloor i/2 \rfloor] \geq A[i]$ for $2 \leq i \leq n$, by heap property; and $A[1]$ contains the largest element. Heapsort uses the heap and its properties most efficiently for sorting[1].

Principle: Based on *selection principle*, and so in-place but not stable (as quicksort). It first constructs the heap from $A[1..n]$ using the procedure BuildHeap (Step 1). It uses two variables: $length[A] = n$ (remains unchanged) and $heapsize[A]$, the latter being decremented as the heap is iteratively reduced in size (Step 2) while swapping the root of (reducing) heap with its last node (Step 3), and then cutting off the last node (current max; Step 4) so as to grow the sorted sublist at the rear end of A. The procedure Heapify is used to rebuild the heap (Step 5), which has lost its *heap property* due to the aforesaid swapping.

[1] J. W. J. Williams. Algorithm 232 – Heapsort, 1964, Communications of the ACM 7(6): 347–348.

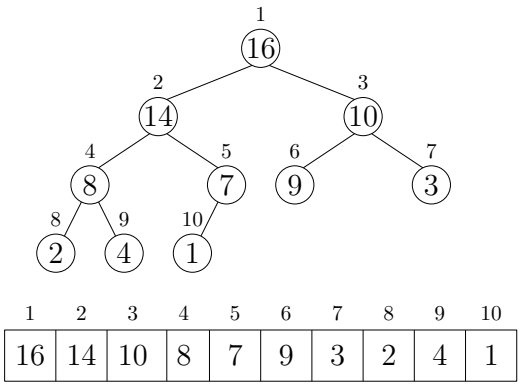


Figure 2.1: Top: A heap as a complete binary tree. Bottom: The corresponding array A.

Algorithm HEAPSORT(A)
1. BUILDHEAP(A)
2. **for** $i \leftarrow length[A]$ **downto** 2
3. SWAP($A[1], A[i]$)
4. $heapsize[A] \leftarrow heapsize[A] - 1$
5. HEAPIFY($A, 1$)

Procedure BUILDHEAP(A)
1. $heapsize[A] \leftarrow length[A]$
2. **for** $i \leftarrow \lfloor length[A]/2 \rfloor$ **downto** 1
3. HEAPIFY(A, i)

Procedure HEAPIFY(A, i)
1. $l \leftarrow$ LEFT$[i]$,
   $r \leftarrow$ RIGHT$[i]$
2. **if** $l \leq heapsize[A] \wedge A[l] > A[i]$
3. $largest \leftarrow l$
4. **else** $largest \leftarrow i$
5. **if** $r \leq heapsize[A] \wedge A[r] > A[largest]$
6. $largest \leftarrow r$
7. **if** $largest \neq i$
8. SWAP($A[i], A[largest]$)
9. HEAPIFY($A, largest$)

2.3.1 Time complexity

The height h of a node in the heap is measured from the bottommost level (height 0) of the heap. Hence, the time required to HEAPIFY a node at height h is $O(h)$, since it involves exchanges between at most h nodes and their left or right children. Now, in a heap having n nodes, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h. Hence, the time complexity of BUILDHEAP is

$$\sum_{h=0}^{\lceil \log n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \right). \tag{2.3}$$

Now, for $|x| < 1$, we have

$$1 + x + x^2 + \ldots + x^h + \ldots = \frac{1}{1-x}.$$

Differentiating and then multiplying both sides by x,

$$x + 2x^2 + 3x^3 + \ldots + hx^h + \ldots = \frac{x}{(1-x)^2},$$

$$\text{or,} \quad \sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2},$$

$$\text{or,} \quad \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 \quad [\text{putting } x = 1/2],$$

$$\text{or,} \quad O\left(n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \right) = O(n). \tag{2.4}$$

Hence, HEAPSORT takes $O(n \log n)$ times, since BUILDHEAP takes $O(n)$ time and each of the $(n-1) = O(n)$ HEAPIFY($A, 1$) calls (Step 5 of HEAPSORT) takes $O(\log n)$ time.

2.4 Linear-time Sorting

All the algorithms discussed in the preceding sections are known as ***comparison sorts***, since they are based on comparison among the input elements. Any comparison sort needs $\Omega(n \log n)$ time in the worst case, as explained next.

A comparison sort can be represented by a (binary) ***decision tree*** in which each non-leaf node corresponds to a comparison between two elements, and each leaf corresponds to a permutation of n input elements. Thus, there are $n!$ leaf nodes, out of which exactly one contains the sorted list. Execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to this node. If the height of the decision tree is h, then it can contain at most 2^h leaves. So,

$$\begin{aligned} & n! \leq 2^h \\ \text{or,} \quad & h \geq \log(n!) > \log\left(\tfrac{n}{e}\right)^n \quad \text{by Stirling's approximation: } n! = \sqrt{2\pi n}\left(\tfrac{n}{e}\right)^n\left(1+\Theta\left(\tfrac{1}{n}\right)\right) \\ \text{or,} \quad & h \geq n\log n - n\log e = \Omega(n\log n). \end{aligned}$$

2.4.1 Counting Sort

Applicable when the input elements belong to a set of size k. It creates an integer array A of size k to count the occurrence of $A[i]$ in the input, and then loops through A to arrange the input elements in order. For $k = O(n)$, the algorithm runs in $O(n)$ time. Counting sort cannot often be used because A needs to be reasonably small for it to be efficient; but the algorithm is extremely fast and demonstrates great asymptotic behavior as n increases. It can also be used to provide stable behavior.

2.4.2 Bucket sort

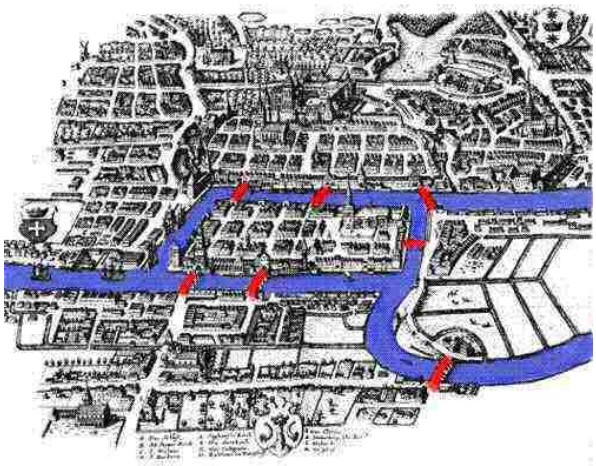
A divide-and-conquer sorting algorithm that generalizes counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm (e.g., insertion sort), or by recursively applying the bucket sort. It is most effective and runs in linear time on data with limited values (e.g., to sort a million integers ranging from 1 to 1000).

2.4.3 Radix sort

Sorts numbers by processing individual digits. It either processes digits of each number starting from the least significant digit (LSD) or from the most significant digit (MSD). The former, for example, first sorts the numbers by their LSD while preserving their relative order using a stable sort; then it sorts them by the next digit, and so on up to their MSD, ending up with the sorted list. Thus, n numbers consisting of k digits can be sorted in $O(nk)$ time, which reduces to $O(n)$ time for a fixed k. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable.

Chapter 3

Graph Algorithms



The story begun in the 18th century, in the quaint town of Königsberg, Prussia, on the banks of the Pregel River. The healthy economy allowed the citizens to build seven bridges across the river. While walking, they created a game for themselves—to walk around the city, crossing each of the seven bridges only once. Even though none of them could invent a route that would allow them to cross each of the bridges only once, still they could not prove that it was impossible. Lucky for them, Königsberg was not too far from St. Petersburg, home of the famous mathematician Leonard Euler.

— History of Mathematics: **On Leonhard Euler (1707-1783)**, *ScienceWeek* (2003).

3.1 Representation of Graphs

Let V be the set of vertices and E be the set of edges. Then the graph $G := (V, E)$ can be represented in either of the following forms.

Adjacency Matrix A two-dimensional array/matrix $((a_{ij}))_{|V|\times|V|}$, where

a_{ij} $= 1$ if $(i, j) \in E$;
$= 0$ otherwise.

Here memory requirement is $O(V^2)$.

Adjacency List An one-dimensional array Adj of size $|V|$, one for each vertex in V. For each $u \in V$, $Adj[u]$ points to each of its adjacent vertices (i.e., $\{v \in V : (u, v) \in E\}$) in linked list implementation. Hence, space complexity is $O(V + E)$.

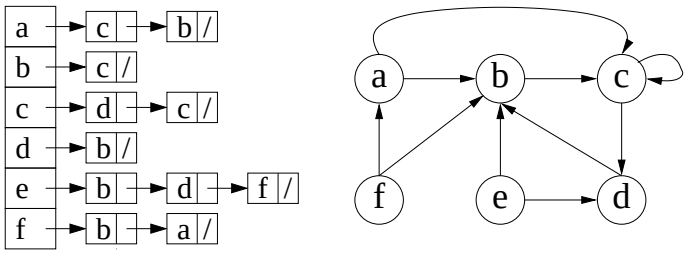


Figure 3.1: Adjacency list (left) of a directed graph (right).

3.2 Breadth First Search (BFS)

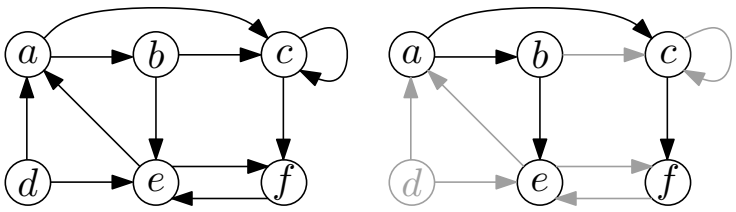
Principle: All vertices at distance k from the start vertex s will be discovered before discovering any vertex at distance $k+1$ (or higher) from s.

input: (i) Graph $G = (V, E)$, directed or undirected.
(ii) Start vertex, $s \in V[G]$.
output: *BFS Predecessor Subgraph* (also called *BFS tree*), $G_\pi = (V_\pi, E_\pi)$, where $E_\pi = \{(v[\pi], v) \ : \ v \in V[G] \ \& \ v[\pi] \neq \text{NIL}\}$.
params: (i) color of $u = c[u] \in \{\text{WHITE}, \text{GRAY}, \text{BLACK}\}$.
(ii) parent of $u = \pi[u]$.
(iii) distance of u from $s = d[u]$.
(iv) queue $= Q$.

Algorithm BFS(G, s)
1. **for** each $u \in V[G]$
2. $\quad c[u] \leftarrow \text{WHITE}$, $\pi[u] \leftarrow \text{NIL}$, $d[u] \leftarrow \infty$
3. $c[s] \leftarrow \text{GRAY}$, $d[s] \leftarrow 0$
4. $Q \leftarrow \{s\}$
5. **while** $Q \neq \emptyset$
6. $\quad u \leftarrow \text{DEQUEUE}(Q, v)$
7. $\quad$ **for** each $v \in Adj[u]$
8. $\quad\quad$ **if** $c[v] = \text{WHITE}$
9. $\quad\quad\quad c[v] \leftarrow \text{GRAY}$, $\pi[v] \leftarrow u$, $d[v] \leftarrow d[u] + 1$
10. $\quad\quad\quad \text{ENQUEUE}(Q, v)$
11. $\quad c[u] \leftarrow \text{BLACK}$

Time complexity:
Steps 1–4: $\Theta(V)$ for initialization. Steps 5–11: $\sum_{u \in V[G]} |Adj(u)| = \Theta(E)$.
So, total time $= \Theta(V + E)$.

Exercise 3.1 The diameter of an undirected graph, G, is defined as the maximum of the shortest path lengths over all vertex pairs of G. Suggest an algorithm and explain its time complexity to find the diameter of G.



Reported output:

v	a	b	c	d	e	f
$\pi[v]$	NIL	a	a	NIL	b	c
$d[v]$	0	1	1	∞	2	2

Figure 3.2: Left: A directed graph, G. Right: DFS Tree with start vertex a; tree vertices and edges are shown in black; the vertex d and the other edges of G which are not in DFS tree, are shown in gray.

3.3 Depth First Search (DFS)

Principle: Traverse as much deep as possible.

input: (i) Graph $G = (V, E)$, directed or undirected.
output: DFS Predecessor Subgraph, $G_\pi = (V, E_\pi)$,
where, $E_\pi = \{(v[\pi], v) \ : \ v \in V[G] \ \& \ v[\pi] \neq \text{NIL}\}$.
params: (i) color of $u = c[u] \in \{\text{WHITE, GRAY, BLACK}\}$.
(ii) parent of $u = \pi[u]$.
(iii) discovery time of $u = d[u]$.
(iv) finishing time of $u = f[u]$.

Algorithm DFS(G)
1. **for** each $u \in V[G]$
2. $c[u] \leftarrow$ WHITE, $\pi[u] \leftarrow$ NIL, $t \leftarrow 0$
3. **for** each $u \in V[G]$
4. **if** $c[u] =$ WHITE
5. DFS_Visit(u, G)

Algorithm DFS_Visit(u, G)
1. $c[u] \leftarrow$ GRAY
2. $t \leftarrow t + 1$
3. $d[u] \leftarrow t$
4. **for** each $v \in Adj[u]$
5. **if** $c[v] =$ WHITE
6. $\pi[v] \leftarrow u$
7. DFS_Visit(v, G)
8. $t \leftarrow t + 1$
9. $f[u] \leftarrow t$
10. $c[u] \leftarrow$ BLACK

Time complexity: Initialization (Steps 1–3) takes $\Theta(V)$ time.
All vertices are visited. That is, finally all vertices are blackened.
Hence, each edge (u, v) is either traversed (i.e. becomes a tree edge T, which occurs if v is WHITE at time $d[u]$) or not traversed (i.e. becomes a non-tree edge $F/B/C$, which occurs if v is not WHITE at time $d[u]$).
Observe that the processing of vertex v for *each edge* (u, v) takes $\Theta(1)$ time. Further, each edge (u, v) is checked exactly once (at time $d[u]$).
Hence, summing up the processing time for v over all (u, v) in $E[G]$, we get $\Theta(E)$.
So, total time $= \Theta(V) + \Theta(E) = \Theta(V + E)$.

Theorem 3.1 [Parenthesis Theorem] *In DFS(G), exactly one of the following three conditions always holds for any pair of vertices u and v.*
(i) the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint;
(ii) the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, and u is a descendant of v in the DFS tree;
(iii) the interval $[d[v], f[v]]$ is contained entirely within the interval $[d[u], f[u]]$, and v is a descendant of u in the DFS tree.

Proof: Follows from different cases and subcases.

Case $d[u] < d[v]$: subcase $d[v] < f[u]$: v was discovered while u was gray, which implies that v is a descendant of u, and therefore, v is finished before u. So $[d[v], f[v]]$ is contained entirely within $[d[u], f[u]]$. (condition (iii))

subcase $f[u] < d[v]$: Here $[d[v], f[v]]$ and $[d[u], f[u]]$ are entirely disjoint. (condition (i))

Case $d[v] < d[u]$: roles of u and v are just reversed.

Theorem 3.2 [White Path Theorem] *In DFS(G), v is a descendant of u if and only if at time $d[u]$, there exists a path from u to v consisting of white vertices only.*

3.3.1 Edge Classification

1. ***Tree edge:*** (u, v) becomes a tree edge iff v was discovered from u by exploring (u, v), i.e., $\pi[v] = u$.
2. ***Back edge:*** (u, v) is a back edge iff v is an ancestor of u; i.e., v is gray at time $d[u]$.
3. ***Forward edge:*** (u, v) is a forward edge iff v is a descendant of u; i.e., $d[u] < d[v] < f[v] < f[u]$ but $\pi[v] \neq u$.
4. ***Cross edge:*** (u, v) is a cross edge iff $d[v] < f[v] < d[u] < f[u]$.

3.4 Topological Sort

A topological sort of a directed acyclic graph ("dag" in short) $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in that ordering. That is, if we redraw G with the topologically sorted vertices on a horizontal line, then each edge is directed from left to right.

Note that, if a directed graph G contains any cycle, then topological sorting of G is not possible.

Algorithm TOPOLOGICAL-SORT(G)
1. DFS(G)
2. insert each vertex $v \in V[G]$ onto the front of a linked list L when it is finished at time $f[v]$
3. **return** L

Time complexity: Since DFS takes $\Theta(V + E)$ time, and insertion of each vertex onto the front of L takes $\Theta(1)$ time, the total time complexity for topological sort is $\Theta(V + E) + |V|\Theta(1) = \Theta(V + E) + \Theta(V) = \Theta(V + E)$.

Lemma 3.1 *A directed graph G is acyclic if and only if DFS(G) yields no back edges.*

Proof: *if* part: Let (u, v) be a back edge in G. Then, v is an ancestor of u in DFS(G). Thus there is a path $v \rightsquigarrow u$, which in combination with (u, v) forms a cycle in G.

only if part: Let c be a cycle is G and v be the first vertex discovered in c. Let (u, v) be the preceding edge in c. Since v is discovered in c before u, there exists a white path $v \rightsquigarrow u$ at time $d[v]$, and so by white-path theorem, u is a descendant of v. Hence, (u, v) becomes a back edge.

Theorem 3.3 TOPOLOGICAL-SORT(G) *produces topological sort of a dag* G.

Proof: Using lemma 3.1.

Exercise 3.2 Find the ordering of vertices produced by topological sorting on the dag G whose adjacency list is as follows.
$a \to c, d$; $b \to d$; $c \to d, g$; $d \to \emptyset$; $e \to \emptyset$; $f \to g, h$; $g \to i$; $h \to i$; $i \to \emptyset$.

Exercise 3.3 Explain whether DFS(G) on an undirected graph $G = (V, E)$ can determine the presence or absence of cycles in G in $O(V)$ time.

Exercise 3.4 Prove or disprove: Topological sort of a dag G may produce more than one ordering of vertices in G.

3.5 Strongly Connected Components

A strongly connected component ("SCC" in short) of a dag $G = (V, E)$ is a maximal set of vertices $V' \subseteq V$ such that for every pair of vertices u and v in V', there exist a path from u to v (i.e., $u \rightsquigarrow v$) and a path from v to u (i.e., $v \rightsquigarrow u$).

Algorithm SCC(G)
1. DFS(G)
2. find $G^T = (V, E^T)$ where $E^T = \{(v, u) : (u, v) \in E\}$
3. rearrange the vertices of G^T to get $G^T = (V^T, E^T)$ such that the vertices in G^T are in decreasing order of their finishing times obtained in step 1
4. DFS(G^T)
5. output the vertices of each DFS tree produced by step 4 as an SCC

Time complexity: Each of step 1 and step 2 takes $\Theta(V + E)$ time. Step 3 needs $\Theta(V)$ time when each vertex is relocated when its DFS finishes in step 1. Step 4 again takes $\Theta(V + E)$ time, thereby giving total time complexity as $\Theta(V + E)$.

Lemma 3.2 *If two vertices are in an SCC, then no path between them ever leaves the SCC.*

Proof: Let u and v belong to the same SCC. Let w be an arbitrary vertex in the path $u \rightsquigarrow w \rightsquigarrow v$. Since there is a path $v \rightsquigarrow u$, we have the path $v \rightsquigarrow u \rightsquigarrow w$, which implies there are paths $w \rightsquigarrow v$ and $v \rightsquigarrow w$. Hence w is in the SCC of v (and of u, there of).

Theorem 3.4 *In any DFS, all vertices in the same SCC are placed in the same DFS tree.*

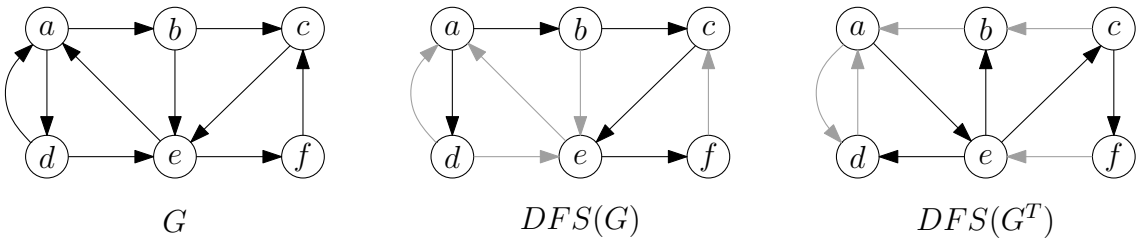


Figure 3.3: The entire directed graph is an SCC.

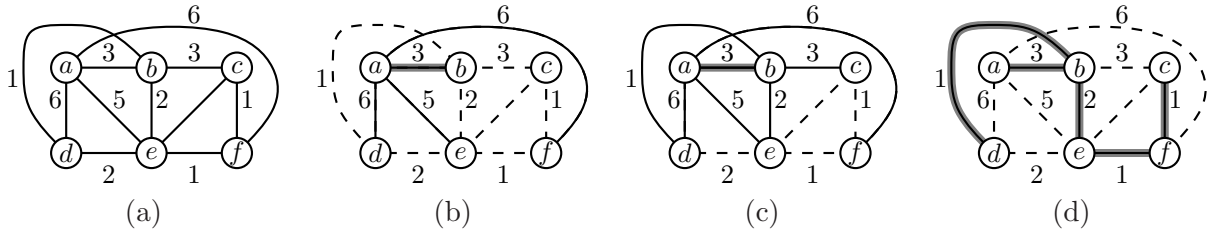


Figure 3.4: Prim's algorithm with $r = a$ (***crossing edges*** as solid lines and ***light edges*** highlighted in gray.) (a) Input graph with edges as dashed lines. (b) After $r = a$ is extracted from Q. (c) After the vertex b is dequeued, since (a, b) is the light edge. (d) Final output with edges of MST highlighted in gray.

Proof: Follows from the white-path theorem. Let r be the vertex that is first discovered (at time $d[r]$) in a strongly connected component $G_S = (V_S, E_S)$. So at time $d[r]$, each vertex $u \in V_S \setminus \{r\}$ is white. Further, there exists a path $r \rightsquigarrow u$ for each $u \in V_S \setminus \{r\}$, and all vertices on $r \rightsquigarrow u$ are white (from Lemma 3.2). Hence, by the white-path theorem, each $u \in V_S \setminus \{r\}$ becomes a descendant of r in the DFS tree rooted at r.

3.6 Minimum Spanning Tree (MST)

A ***spanning tree*** of any connected, undirected graph $G(V, E)$ is any tree $T(V, E')$ such that $E' \subseteq E$ (Fig. 3.6.1). An ***MST*** is defined for a connected, undirected, weighted graph $G(V, E, w)$. In a ***weighted graph***, each edge $(u, v) \in E$ has a weight $w(u, v)$, which is any (positive or negative) number. An MST of G is such a spanning tree of G for which the sum of weights of its edges is minimal. There are several ***greedy algorithms*** to find an MST of a graph G. A few are explained next.

3.6.1 MST by Kruskal's Algorithm

Principle: It uses the following greedy idea. At a certain iteration when the algorithm is in action, if there are k trees, $T_1, T_2, \ldots, T_k$, such that $V[T_1] \cup V[T_2] \cup \ldots \cup V[T_k] = V$, and each T_i (local optimum) is a sub-tree of the MST (global optimum) [that's greedy!], then in the next iteration we choose the ***light edge*** whose two nodes belong to two different sub-trees, say T_i and T_j $(1 \leq i, j \leq k)$, and whose weight is minimal in $E \setminus (E[T_1] \cup E[T_2] \cup \ldots \cup E[T_k])$. Hence, in the next iteration, T_i and T_j form a larger sub-tree (a larger local optimum, thereof) of MST. The algorithm thus goes edge by edge, in non-decreasing weights of their edges, and accepts an edge (a light edge) only if its weight is the current minimum and it does not form a cycle.

Time Complexity: Preprocessing needs sorting the edge weights, requiring $O(E \log E)$ time. Whether an edge (u, v) having the current minimal weight in each iteration uses FIND-SET operation on u and v, requiring $\alpha(E, V) = O(\log E)$ time per edge, i.e., $E \cdot O(\log E) = O(E \log E)$ in total. Hence, total time complexity of Kruskal's MST algorithm is $O(E \log E)$.

3.6.2 MST by Prim's Algorithm

Principle: It uses a single local optimum (sub-solution), and grows it "greedily" by including one light edge in each iteration, until the global optimum (MST) is obtained. Trivially, the local sub-solution is initialized by an arbitrary vertex, r.

Algorithm MST-PRIM(G, r)
1. $Q \leftarrow V$ $\triangleright$ Q is a min-heap based on key
2. **for** each $u \in Q$
3. $\quad key[u] \leftarrow \infty$
4. $key[r] \leftarrow 0, \pi[r] \leftarrow$ NIL
5. **while** $Q \neq \emptyset$
6. $\quad u \leftarrow$ EXTRACT-MIN(Q) $\triangleright$ and re-heapify $Q \Rightarrow V \cdot O(\log V) = O(V \log V)$
7. $\quad\quad$ **for** each $v \in Adj[u]$
8. $\quad\quad\quad$ **if** $v \in Q$ and $w(u, v) < key[v]$
9. $\quad\quad\quad\quad key[v] \leftarrow w(u, v), \pi[v] \leftarrow u \Rightarrow E \cdot O(\log V) = O(E \log V)$

Implementation: For each vertex $u \in V$, keep a boolean flag that indicates whether or not u is in Q (for Step 8). Also maintain another variable to know the index of u in Q, which is required to update its key and π whenever Step 9 is executed.

Time Complexity: Initialization (Steps 1–4) takes $O(V)$ time. In Step 6, EXTRACT-MIN(Q) needs $O(\log V)$ time each time until Q is empty after V iterations, thus requiring $O(V \log V)$ time in total. Checking for Step 8 is called exactly E times; each check requiring $O(1)$ time with the aforesaid implementation. If the **if** condition is true, then the update is done by Step 8 in $O(\log V)$ time, as Q is a heap. Step 8 is called $O(E)$ times and Step 9 at most $O(E)$ times, hence giving a total time complexity of $O(E \log V)$. The resultant time complexity of the entire algorithm is, therefore, $O(V) + O(V \log V) + O(E \log V) = O(E \log V)$.

3.6.3 Elements of Greedy Algorithms

Greedy choice: A globally optimal solution can be obtained by making a locally optimal (greedy) choice.
Example: In Kruskal's and Prim's algorithms, the choice of light edge.

Optimal substructure: An optimal solution to the problem contains optimal solutions to subproblems.
Example: Let's choose any subgraph $G'(V', E') \subset G(V, E)$ such that $(u, v) \in E'$ if and only if $u \in V'$, $v \in V'$, and $(u, v) \in E$. Then the MST of G' is a sub-tree of some MST of G.

Chapter 4

Dynamic Programming

If you optimize everything, you will always be unhappy.

— **Donald Knuth**

4.1 Elements of Dynamic Programming

Optimal substructure: An optimal solution to the problem contains optimal solutions to subproblems (as in greedy algorithms).

Overlapping subproblems: A sub-subproblem P' has to be solved repeatedly to solve two or more subproblems whose intersection is P'. Hence, the problem contains a recursive nature, which often creates an illusion of exponential possibilities.

Example problems: Matrix-chain multiplication; Longest common subsequence; 0-1 Knapsack problem; Optimal triangulation of a convex polygon (each triangle has some weight, e.g., $w(a,b,c) = |ab| + |bc| + |ca|$); Denomination problem.

4.2 Matrix-chain Multiplication

Given a chain of matrices $A_1 A_2 \cdots A_n$, the problem is to obtain its fully ***parenthesized form*** such that the number of scalar multiplications is minimized. Apparently, the problem seems to be have exponential complexity as it relates to ***Catalan number***, as follows.

Number of parenthesizations:

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 = \textbf{\textit{Catalan number}}\ C(n-1) \end{cases}$$

$$\text{where} \quad C(n) = \frac{1}{n+1}\binom{n}{2n} = \Omega\left(\frac{4^n}{n^{3/2}}\right). \tag{4.1}$$

Using DP, we can solve it in low-order polynomial time. The DP-based algorithm is built on the following observation, which explains the two elements of DP.

Optimal substructure & overlapping subproblems: Parenthesization of $A_1A_2\cdots A_k$ $(k<n)$ within the optimal parenthesization of $A_1A_2\cdots A_n$ must be an optimal parenthesization of $A_1A_2\cdots A_k$; and similar for $A_{k+1}\cdots A_n$.

Thus, the minimum number of scalar multiplications for $A_iA_{i+1}\ldots A_j$ is

$$m[i,j]=\begin{cases}0 & \text{if } i=j,\\ \min\limits_{i\le k<j}(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j) & \text{if } i<j;\end{cases}$$

where each A_i has p_{i-1} rows and p_i columns.

Example: Minimize the number of scalar multiplications to compute $A_1A_2\cdots A_6$ for:

matrix	dimension
A_1	20×30
A_2	30×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

In the given problem, $j-i=1,2,\ldots,5$.
We start with $j-i=1$ and finally end at $j-i=5$ to reach the final solution, i.e., $m[1,6]$.

1. $j=1$:
   (a) $m[1,2]=p_0p_1p_2=20\times 30\times 15=9000$
   (b) $m[2,3]=p_1p_2p_3=30\times 15\times 5=2250$
   (c) $m[3,4]=p_2p_3p_4=15\times 5\times 10=750$
   (d) $m[4,5]=p_3p_4p_5=5\times 10\times 20=1000$
   (e) $m[5,6]=p_4p_5p_6=10\times 20\times 25=5000$
2. $j=2$:
   (a) $m[1,3]=\min\left\{\begin{array}{lll} m[1,2]+p_0p_2p_3 & =9000+1500 & =10500\\ m[2,3]+p_0p_1p_3 & =2250+3000 & =5250\end{array}\right\}=5250(k=1)$
   (b) $m[2,4]=\min\left\{\begin{array}{lll} m[2,3]+p_1p_3p_4 & =2250+1500 & =3750\\ m[3,4]+p_1p_2p_4 & =750+4500 & =5250\end{array}\right\}=3750(k=3)$
   (c) $m[3,5]=\min\left\{\begin{array}{lll} m[3,4]+p_2p_4p_5 & =750+3000 & =3750\\ m[4,5]+p_2p_3p_5 & =1000+1500 & =2500\end{array}\right\}=2500(k=3)$
   (d) $m[4,6]=\min\left\{\begin{array}{lll} m[4,5]+p_3p_5p_6 & =1000+2500 & =3500\\ m[5,6]+p_3p_4p_6 & =5000+1250 & =6250\end{array}\right\}=3500(k=5)$
3. $j=3$:
   (a) $m[1,4]=\min\left\{\begin{array}{lll} m[1,3]+p_0p_3p_4 & =5250+1000 & =6250\\ m[2,4]+p_0p_1p_4 & =3750+6000 & =9750\\ m[1,2]+m[3,4]+p_0p_2p_4 & =9000+750+3000 & =12750\end{array}\right\}=6250(k=3)$
   (b) $m[2,5]=\min\left\{\begin{array}{lll} m[2,4]+p_1p_4p_5 & =3750+6000 & =9750\\ m[3,5]+p_1p_2p_5 & =2500+9000 & =11500\\ m[2,3]+m[4,5]+p_1p_3p_5 & =2250+1000+3000 & =6250\end{array}\right\}=6250(k=3)$
   (c) $m[3,6]=\min\left\{\begin{array}{lll} m[3,5]+p_2p_5p_6 & =2500+7500 & =10000\\ m[4,6]+p_2p_3p_6 & =3500+1875 & =5375\\ m[3,4]+m[5,6]+p_2p_4p_6 & =750+5000+\times & =\times\end{array}\right\}=5375(k=3)$

4. $j = 4$:

(a) $m[1,5] = \min \left\{ \begin{array}{lll} m[1,4] + p_0p_4p_5 & = 6250 + 4000 & = 10250 \\ m[2,5] + p_0p_1p_5 & = 6250 + 12000 & = 18250 \\ m[1,3] + m[4,5] + p_0p_3p_5 & = 5250 + 1000 + 2000 & = 8250 \\ m[1,2] + m[3,5] + p_0p_2p_5 & = 9000 + 2500 + 6000 & = 17500 \end{array} \right\} = 8250(k = 3)$

(b) $m[2,6] = \min \left\{ \begin{array}{lll} m[2,5] + p_1p_5p_6 & = 6250 + 15000 & = 21250 \\ m[3,6] + p_1p_2p_6 & = 5375 + 11250 & = 16625 \\ m[2,3] + m[4,6] + p_1p_3p_6 & = 2250 + 3750 + 3500 & = 9500 \\ m[2,4] + m[5,6] + p_1p_4p_6 & = 3750 + 5000 + 7500 & = 16250 \end{array} \right\} = 9500(k = 3)$

5. $j = 5$:

(a) $m[1,6] = \min \left\{ \begin{array}{lll} m[1,5] + p_0p_5p_6 & = 8250 + 10000 & = 18250 \\ m[2,6] + p_0p_1p_6 & = 9500 + 15000 & = 24500 \\ m[1,2] + m[3,6] + p_0p_2p_6 & = 9000 + 5375 + 7500 & = 21875 \\ m[1,3] + m[4,6] + p_0p_3p_6 & = 5250 + 3500 + 2500 & = 11250 \\ m[1,4] + m[5,6] + p_0p_4p_6 & = 6250 + 5000 + 5000 & = 16250 \end{array} \right\} = 11250(k = 3)$

Table (value of k is shown parenthesized with the corresponding m):

	1	2	3	4	5	6
1	0	9000 (1)	5250 (1)	6250 (3)	8250 (3)	11250 (3)
2		0	2250 (2)	3750 (3)	6250 (3)	9500 (3)
3			0	750 (3)	2500 (3)	5375 (3)
4				0	1000 (4)	3500 (5)
5					0	5000 (5)
6						0

Since $k[1,6] = 3$, we have the parenthesization as follows: $(A_1A_2A_3)(A_4A_5A_6)$
$= (A_1(A_2A_3))((A_4A_5)A_6)$, since $k[1,3] = 1$ and $k[4,6] = 5$.

4.3 Longest common subsequence (LCS)

Given two sequences, $A = \langle a_i : i = 1, \ldots, m\rangle$ and $\langle b_j : j = 1, \ldots, n\rangle$, find the longest sequence $C = \langle c_k : k = 1, \ldots, p\rangle$ such that

(i) each c_k matches with some $a \in A$ and with some $b \in B$;

(ii) all the characters (if any) of C preceding c_k match with some $k-1$ characters of A preceding a and with some $k-1$ characters of B preceding b.

$C_k = LCS(A_m, B_n)$ (or, simply $C = LCS(A, B)$) is called the ***longest common subsequence*** of A and B.

Example: $A = \langle$algorithms$\rangle$, $B = \langle$allgorhythm$\rangle$: $C = \langle$`algorthm`$\rangle$.

Optimal substructure: The observation is as follows.

(i) if $a_m = b_n$, then $c_k = a_m = b_n$ and $C_{k-1} = LCS(A_{m-1}, B_{n-1})$.

(ii) if $a_m \neq b_n$, then $z_k \neq x_m$ implies $C_k = LCS(A_{m-1}, B_n)$; else $z_k \neq y_n$ implying $C_k = LCS(A_m, B_{n-1})$.

Overlapping subproblems: Let $|LCS(A_i, B_j)|$ be the length of $LCS(A_i, B_j)$. Then we have

$$|LCS(A_i, B_j)| = \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ |LCS(A_{i-1}, B_{j-1})| + 1 & \text{if } i > 0 \wedge j > 0 \wedge x_i = y_j \\ \max\left(|LCS(A_i, B_{j-1})|, |LCS(A_{i-1}, B_j)|\right) & \text{if } i > 0 \wedge j > 0 \wedge x_i \neq y_j \end{cases} \quad (4.2)$$

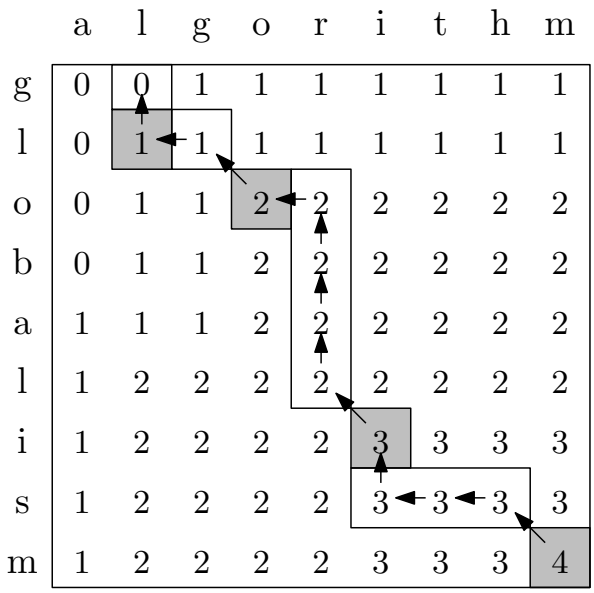


Figure 4.1: Approximate string matching between two strings, 'algorithm' and 'globalism'.

4.4 0-1 (Binary) Knapsack Problem

Given a set $A = \{a_i : i = 1, 2, \ldots, n\}$ where each a_i has benefit b_i and (a positive integer) weight w_i. Given a knapsack whose weight carrying capacity is $w_{\max}$. The problem is to put some or all items from A into the knapsack without exceeding its capacity such that the total benefit is maximized.

Let $A_i = \{a_1, a_2, \ldots, a_i\}, 1 \leq i \leq n$.
Let $B[i, w]$ = maximum total benefit over all possible subsets $A'_i \subseteq A_i$ such that $\sum_{a_j \in A'_i} w_j = w$.

Observation: If $w_i > w$, then a_i can never be in the maximum-benefit solution A'_i corresponding to $B[i, w]$. Otherwise, there may arise two cases: In case $a_i \in A'_i$, we have $B[i, w] = B[i-1, w-w_i] + b_i$; and in case $a_i \notin A'_i$, we have $B[i, w] = B[i-1, w]$. Thus,

$$B[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ B[i-1,w] & \text{if } w_i > w \\ \max\{B[i-1,w], B[i-1,w-w_i] + b_i\} & \text{if } w_i \leq w. \end{cases} \tag{4.3}$$

The elements of DP are evident from Eqn. 4.3. In the resultant algorithm, $B[w]$ contains the maximum benefit of items with total weight *at most* w for the current value of i.

Algorithm 0-1-KnapSack
01. **for** $w \leftarrow 0$ to $w_{\max}$
02. $\quad B[w] \leftarrow 0$
03. **for** $i \leftarrow 1$ to n
04. $\quad$ **for** $w \leftarrow w_{\max}$ downto w_i ▷ note the loop order
05. $\quad\quad$ **if** $B[w] < B[w - w_i] + b_i$
06. $\quad\quad\quad B[w] \leftarrow B[w - w_i] + b_i$
07. **return** $B[w_{\max}]$

Time complexity: Steps 3–6 dominate over other steps in runtime. The inner **for** loop (Steps 4–6) takes $O(w_{\max})$ time and the outer **for** loop (Step 3) iterates for n times. Hence, the overall time complexity is $O(n) \times O(w_{\max}) = O(nw_{\max})$.[1]

Example: Let $n = 4$, $w_{\max} = 6$.

i	1	2	3	4
b_i	7	9	6	8
w_i	2	4	7	3

Iterations:

	w	0	1	2	3	4	5	6
$B[w]$	initial	0	0	0	0	0	0	0
	$i = 1$	0	0	7	7	7	7	7
	$i = 2$	0	0	7	7	9	9	16
	$i = 3$	0	0	7	7	9	9	16
	$i = 4$	0	0	0	8	9	15	16

$\Rightarrow$ max benefit = 16.

4.5 Fractional Knapsack Problem

If we are allowed to take a fraction of any item $a_i \in A$ while maximizing the total benefit without exceeding the knapsack capacity, then it's fractional knapsack problem. Such a problem is solvable by ***greedy approach*** as follows. Compute unit-benefit $c_i = b_i/w_i$ for each a_i, and arrange a_i's in decreasing/non-increasing order of c_i's. Now, select the items starting from the first item of the sorted list, ending with a fraction f_j of some item, a_j, such that all other items selected so far (possibly excepting a_j, if $f_j < 1$) are fully selected. Due to sorting, its time complexity $= O(n \log n)$.

Clearly, fractional knapsack problem is much easier than 0-1 knapsack.

Example: Let $n = 4$, $w_{\max} = 6$. We arrange the items in decreasing order of c_i's as follows.

i	1	2	3	4
b_i	12	12	6	7
w_i	2	3	2	7
c_i	6	4	3	1

Output: $f_1 = 1, f_2 = 1, f_3 = 1/2, f_4 = 0$, and total benefit $= 12 + 12 + 3 = 27$.

[1] Such an algorithm is called a ***pseudo-polynomial algorithm***, since its polynomial time complexity is not only dependent on the input size, i.e., n, but also on $w_{\max}$, which is just an input value. In fact, 0-1 knapsack problem is considered as an ***NP-complete problem***.

4.6 Exercise Problems

4.6.1 Maximum-Value Contiguous Subsequence

Given a sequence of n (positive or negative) real numbers $A[1..n]$, determine a contiguous subsequence $A[i..j]$ for which the sum of elements in the subsequence is maximized.

4.6.2 Making Change

Given n types of coin denominations of values $1 = v_1 < v_2 < \ldots < v_n$, give an algorithm which makes change for an amount of money x with as few coins as possible.

4.6.3 Longest Increasing Subsequence

Given a sequence of n real numbers $A[1..n]$, determine a strictly increasing subsequence (not necessarily contiguous) of maximal length.

4.6.4 Building Bridges

A country has a straight horizontal river passing through it. There are n cities on one bank with same y-coordinates and x-coordinates as $a_1 < a_2 < \ldots < a_n$, and n cities on the other bank with same y-coordinates and x-coordinates as $b_1 < b_2 < \ldots < b_n$. Connect as many pairs of cities (each pair with two cities from two banks) as possible with bridges such that no two bridges cross. Report the number of bridges.

4.6.5 Sum-based Balanced Partition

Given a set $S = \{s_1, \ldots, s_n\}$ of n integers (need not be distinct), each in the range $[0, k]$. Partition S into two subsets S_1 and S_2 such that the difference between the sum of elements of S_1 and the sum of elements of S_2 is minimized.

4.6.6 Optimal Game Strategy

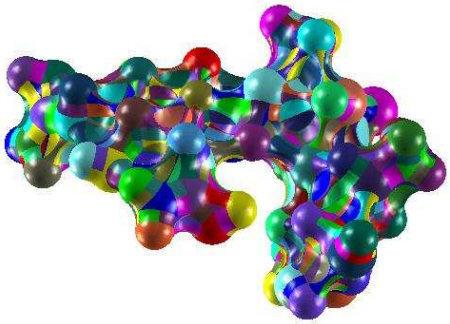
Consider a row of n coins of values $v_1, \ldots, v_n$, where n is even. We play a game against an opponent by alternating turns. In each turn, a player picks either the first or the last coin from the row. Determine the maximum possible amount of money we can definitely win if we move first.

4.6.7 Two-Person City Traversal

Given an ordered sequence of n cities and all the $\binom{n}{2}$ distances for all pairs of cities. Partition the sequence of cities into two subsequences (not necessarily contiguous) such that if Person A visits all cities in the first subsequence (in order) and Person B visits all cities in the second subsequence (in order), then the sum of the total distances traveled by A and B is minimized.

Chapter 5

Geometric Algorithms



We could present spatially an atomic fact which contradicted the laws of physics, but not one which contradicted the laws of geometry.

— Ludwig Wittgenstein
Tractatus Logico Philosophicus, *New York* (1922).

5.1 Closest Pair

Given a set of points $P = \{p_1, \ldots, p_n\}$, the problem is to find the minimum distance $\delta_{\min}$ between two points of P. The naive algorithm is to consider all $\binom{n}{2}$ point-pairs of P, compute their distances, and find the minimum of these $\binom{n}{2} = O(n^2)$ distances, which needs $O(n^2)$ time. An efficient algorithm based on divide-and-conquer approach [M.I. Shamos and D. Hoey. Closest-point problems. *Proc. FOCS*, pp. 151–162, 1975.] is given below, which needs $O(n \log n)$ time.

5.1.1 Divide-and-Conquer Algorithm

1. Sort the points of P to P_x and to P_y using x- and y-coordinates, respectively. (This step is out of recursion.)
2. Partition P into P_L and P_R by the vertical median-line $l_\mu : x = x_\mu$, using P_x.
3. Recursively compute the minimum distances δ_L and δ_R for P_L and P_R, respectively.
4. $\delta' \leftarrow \delta \leftarrow \min(\delta_L, \delta_R)$.
5. Traverse P_y and *append* a point $(x, y) \in P_y$ to Q_y (initialized as empty) if $x \in [x_\mu - \delta, x_\mu + \delta]$. If $x \in [x_\mu - \delta, x_\mu]$, then mark it as GRAY; otherwise WHITE (Fig. 5.1b).

   Result: $Q_y (= Q_L \cup Q_R)$ is y-sorted.
6. Traverse Q_y in order from first to last, and for each GRAY point $p_L \in Q_y$,
   (a) compute the distances of *four* BLACK points following p_L and *four* BLACK points preceding p_L in Q_y;
   (b) find the minimum δ'' of the above eight distances;
   (c) $\delta' \leftarrow \min(\delta', \delta'')$.
7. *return* $\min(\delta, \delta')$.

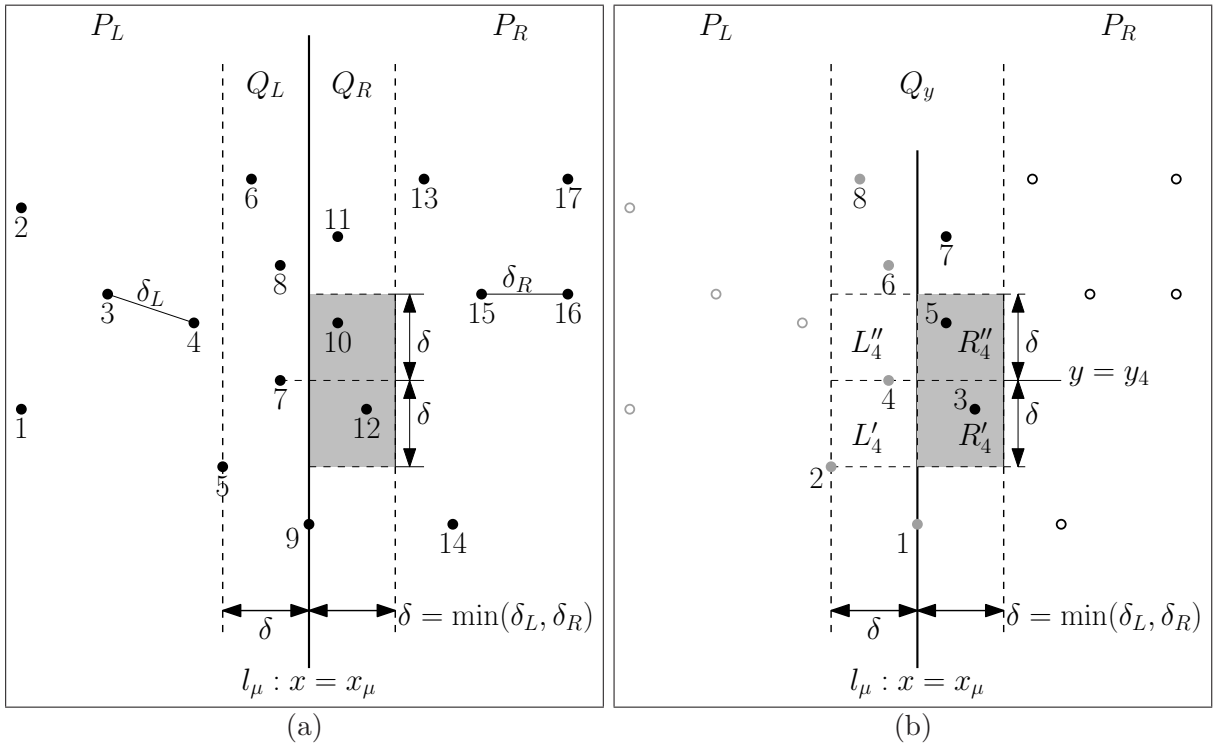


Figure 5.1: Demonstration on a small point set P having 17 points. (a) Strips $Q_L \subseteq P_L$ and $Q_R \subseteq P_R$. (b) Points and their indices in $Q_y = Q_L \cup Q_R$, sorted on y-coordinates. Points from Q_L are stored as GRAY points and those from Q_R as BLACK points, in Q_y.

Time complexity

Step 1: $O(n \log n)$, Step 2: $O(1)$, Step 3: $2 \times T(n/2)$, Step 4: $O(1)$, Step 5: $O(n)$.

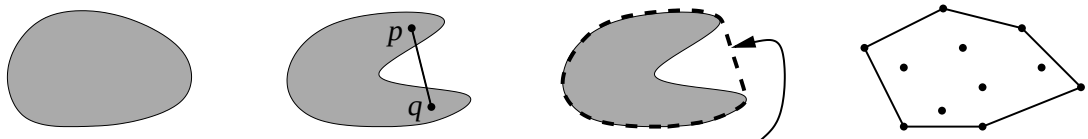
Step 6(a): The four BLACK points following p_L in Q_y would lie in the $\delta \times \delta$ square box, namely R'', or/and above. (In Fig. 5.1b, there is one BLACK point in R''_4 corresponding to Point 4.) Also, there can be at most three other GRAY points in the $\delta \times \delta$ square box L'' containing p_L. (In Fig. 5.1b, there is no other GRAY point in L''_4.)

In effect, there can be at most seven other (three GRAY and four BLACK) points in $R'' \cup L''$. Hence, we have to *see at most seven points* following p_L to compute the distances of *four* BLACK points following p_L in Q_y.

Similar arguments apply also for computing the distances of *four* BLACK points preceding p_L in Q_y.

So, Step 6(a) (and hence Steps 6(b) and 6(c)) needs $O(1)$ time for each GRAY point $p_L \in Q_y$, thereby requiring $O(n)$ time for all GRAY points of Q_y.

Hence, for Steps 2–7, the time complexity is $T'(n) = 2T'(n/2) + O(n) = O(n \log n)$, which, when added with the time complexity of sorting (Step 1), gives the overall time complexity as $\boxed{T(n) = T'(n) + O(n \log n) = O(n \log n)}$.



(a) convex set (b) non−convex set (c) convex hull (d) convex hull of a point set

Figure 5.2: Set shown in (a) is a convex set as the line segment joining any two points of the set lies entirely inside the set. In (b), a part of the line segment $\overline{pq}$ lies outside the set, and so the set is not convex. Shown in (c) is the convex hull of the set in (b). In (d), a point set and its convex hull have been shown.

5.2 Convex Hull

Definition 5.1 (convex set) *A set S is convex if and only if for any pair of points $p, q \in S$, the line segment $\overline{pq}$ is completely contained in S.*

Examples of convex sets: circles; ellipses; triangles; squares; rectangles; trapeziums, a polygon having each internal angle $< 180^0$.
Examples of non-convex sets: a polygon having some internal angle(s) $> 180^0$; dumbbells; star-shaped figures; (2D projections of) most of the real-world objects like chairs, tables, tee-shirts, human body, etc.

Definition 5.2 (convex polygon) *A polygon P whose* any *diagonal lies entirely inside P is a convex polygon. To elaborate, a polygon P is convex if and only if the line segment pq connecting* any *two points p and q lying on the boundary of P, lies entirely inside P.*

Definition 5.3 (convex hull) *The smallest convex set $\mathcal{C}(A)$ that contains a set A is called the convex hull of A.*

Observation 5.1 $\mathcal{C}(A)$ is the intersection of all convex sets that contain A.

Observation 5.2 The convex hull $\mathcal{C}(P)$ of an arbitrary polygon P is given by the convex hull of the vertices of P.

5.2.1 Convex hull of a point set

Let $P = \{p_1, p_2, \ldots, p_n\}$ be a point set in 2D plane. Then, we have the following observations on $\mathcal{C}(P)$.

Observation 5.3 $\mathcal{C}(P)$ is a convex polygon and each vertex of $\mathcal{C}(P)$ is a point in P (i.e., $\mathcal{C}(P) \subseteq P$).

Observation 5.4 During clockwise traversal of $\mathcal{C}(P)$, if a vertex $p \in \mathcal{C}(P)$ is immediately followed by the vertex $q \in \mathcal{C}(P)$, then each point of $P \setminus \{p, q\}$ lies either on the right side of or on the line segment $\overline{pq}$.

5.2.2 Naive algorithm of convex hull

Based on Observation 5.4, the naive algorithm for finding $\mathcal{C}(P)$ of a given point set P is developed. Here, E denotes the set of edges of $\mathcal{C}(P)$.

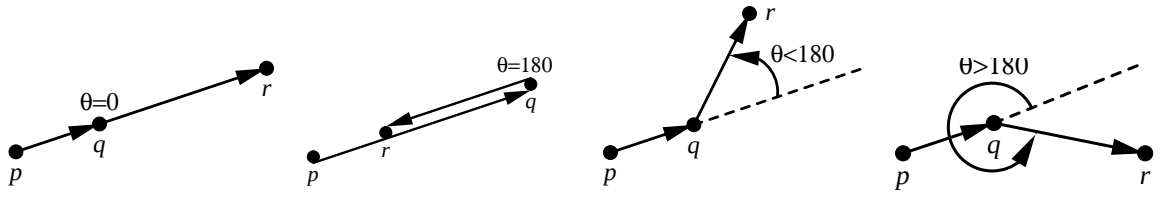
Algorithm 5.1 CONVEX-HULL-NAIVE(P)

```
01. E ← ∅
02. for each p ∈ P
03.     for each q ∈ P \ {p}
04.         flag ← TRUE
05.         for each r ∈ P \ {p, q}
06.             if D(p, q, r) > 0  ▷ (p, q, r) is a left turn
07.                 flag ← FALSE
08.                 break
09.         if flag = TRUE
10.             E ← E ∪ (p, q)
11.             break
12. process E to obtain the vertices of C(P) in clockwise order
```

Computation of $D(p,q,r)$ in step 6: Let $p=(x_p,y_p)$, $q=(x_q,y_q)$, and $r=(x_r,y_r)$. Then the vector from p to q is given by $(x_q-x_p)\widehat{i}+(y_q-y_p)\widehat{j}$, where $\widehat{i}$ and $\widehat{j}$ represent the respective unit vectors along x-axis and y-axis; and similarly, $\overrightarrow{qr}=(x_r-x_q)\widehat{i}+(y_r-y_q)\widehat{j}$. Thus,

$$\begin{aligned}
\overrightarrow{pq}\times\overrightarrow{qr} &= \big((x_q-x_p)\widehat{i}+(y_q-y_p)\widehat{j}\big)\times\big((x_r-x_q)\widehat{i}+(y_r-y_q)\widehat{j}\big)\\
&= \big((x_q-x_p)(y_r-y_q)-(y_q-y_p)(x_r-x_q)\big)\widehat{k},\ \text{since } \widehat{i}\times\widehat{j}=\widehat{k}, \widehat{j}\times\widehat{i}=-\widehat{k}\\
&= \big(x_qy_r-x_qy_q-x_py_r+x_py_q-y_qx_r+y_qx_q+y_px_r-y_px_q\big)\widehat{k}\\
&= \big(x_qy_r-x_py_r+x_py_q-y_qx_r+y_px_r-y_px_q\big)\widehat{k}\\
&= \big(x_p(y_q-y_r)+x_q(y_r-y_p)+x_r(y_p-y_q)\big)\widehat{k}\\
&= D\widehat{k},
\end{aligned}$$

where $D=\big(x_p(y_q-y_r)+x_q(y_r-y_p)+x_r(y_p-y_q)\big)$. Hence, if $D>0$, then the vector $D\widehat{k}$ is directed towards the positive z-axis (i.e., outward from the plane of paper as shown in Fig. 5.3), and indicates a left turn; whereas, $D=0$ indicates no turn, and $D<0$ indicates a right turn at q w.r.t. p as its previous point and q as its next point.



(a) $D=0$ (r lies on $\overrightarrow{pq}$ and in front of q). (b) $D=0$ (r lies on $\overrightarrow{pq}$ and to the back of q). (c) $D>0$ (r lies left of $\overrightarrow{pq}$). (d) $D<0$ (r lies right of $\overrightarrow{pq}$).

Figure 5.3: Determining the position of the point r w.r.t. the vector directed from p to q.

Time complexity: Step 1 executes for n times; for each iteration of step 1, step 2 loops at most $n-1$ times; and for each iteration of step 2, step 3 loops at most $n-2$ times. So the total time complexity from step 1 to step 11 is bounded by $O(n^3)$.

While processing E, we remove the first edge, say (q_1, q_2), from E, and include q_1 (as first vertex) and q_2 (as second vertex) in (the vertex set of) $\mathcal{C}(P)$. Next, we search in E for the edge with q_2 as its first vertex; let the corresponding edge in E be (q_2, q_3). We remove (q_2, q_3) from E and include q_3 (i.e., append as third vertex) in $\mathcal{C}(P)$. We continue this process in step 12 until E is empty. If h be the number of vertices of $\mathcal{C}(P)$, then there would be $h(\leq n)$ edges in E. Hence time complexity of step 12 is dominated by the search operations in E, which would be at most $(h-1)+(h-2)+\ldots+1 = O(h^2)$, which is $O(n^2)$ in the worst case.

So, the worst case total time complexity of the algorithm Convex-Hull-Naive(P) is $O(n^3)$.

5.2.3 An incremental algorithm (Graham scan)

Based on the following observation:

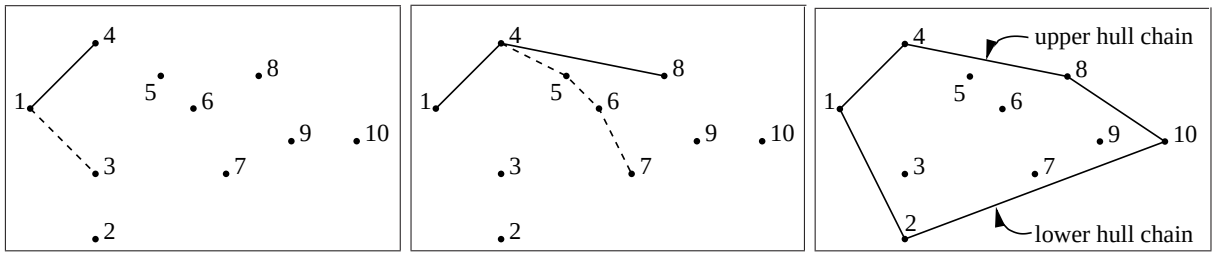
Observation 5.5 If p'_1 and p'_n be the leftmost vertex and the rightmost vertex of P, then p'_1 and p'_n must be the (leftmost and rightmost) vertices of $\mathcal{C}(P)$. [If there are multiple leftmost vertices (each with minimum x-coordinate), then we consider the one with minimum y-coordinate as the actual leftmost vertex. Similarly, in case of multiple rightmost vertices, we consider the vertex with (maximum x and) maximum y as the actual rightmost vertex.]

In Graham scan algorithm, we compute the upper hull chain starting from p'_1 and ending at p'_n considering the vertices one by one according to their non-decreasing x-coordinates, and store the upper hull vertices in an ordered list, U. The lower hull chain is derived and stored in the list L in a similar fashion starting from p'_n and ending at p'_1 considering the non-increasing x-coordinates of the points of P. Since there may be two or more vertices with same x-coordinate, the vertices are lexicographically sorted considering x-coordinate as the primary value and y-coordinate as the secondary value. In the sorted list, namely $P' = \{p'_1, p'_2, \ldots, p'_{n-1}, p'_n\}$, for each vertex $p'_i := (x'_i, y'_i)$ and its following vertex p'_{i+1}, we have

$$\begin{array}{ll} \text{either} & x'_i < x'_{i+1} \\ \text{or} & x'_i = x'_{i+1} \text{ and } y'_i < y'_{i+1}. \end{array}$$

Time complexity: The operations on U in step 4 and step 6 are analogous to push and pop operations respectively that are defined on a stack. For each iteration of the **for** loop in step 3, a vertex p'_i is pushed to U exactly once. Consideration of the initial two pushes (vertices p'_1 and p'_2) in step 2, therefore, gives the total number of push operations as $n = O(n)$. Maximum number of pops is $n-2$; since total number of pops never exceeds total number of pushes in a stack, and no pop in step 6 is performed if there are 2 vertices in U (step 5). Note that, if any pop is performed in step 6 (once the two conditions in step 5 are satisfied), the program flow moves back to step 5. The two tests in step 5 are performed at least once for each iteration of the **for** loop, and in addition, they are performed once after each pop operation in step 6. Effectively, in total, the tests in step 5 are performed for at most $2(n-2) = O(n)$ times. Hence the time complexity for finding the upper hull vertices needs $O(n)$ time.

A similar explanation gives $O(n)$ time for computing the lower hull. Thus, the total time complexity, considering that the lexicographic sorting in step 1 needs $O(n \log n)$ time, of Graham scan algorithm is given by $O(n \log n)$.



(a) $U = \{p_1, p_4\}$ (b) $U = \{p_1, p_4, p_8\}$ (c) $U = \{p_1, p_4, p_8, p_{10}\}$, $L = \{p_{10}, p_2, p_1\}$

Figure 5.4: Demonstration of Graham scan algorithm on a small point set, the points being labeled by their indices for simplicity.

Algorithm 5.2 CONVEX-HULL-GRAHAM-SCAN(P)

01. lexicographically sort (x, y) of the points in P to get P'
02. $U \leftarrow \{p'_1, p'_2\}$
03. **for** $i \leftarrow 3$ to n
04. $\quad U \leftarrow \{p'_i\}$
05. $\quad$ **while** U contains at least 3 vertices **and** the last three vertices in U do not make a right turn
06. $\quad\quad$ remove the last but one vertex from U
07. $L \leftarrow \{p'_n, p'_{n-1}\}$
08. **for** $i \leftarrow n - 2$ to 1
09. $\quad U \leftarrow \{p'_i\}$
10. $\quad$ **while** L contains at least 3 vertices **and** the last three vertices in L do not make a right turn
11. $\quad\quad$ remove the last but one vertex from L
12. remove p_n and p_1 from L and append L with U to get $\mathcal{C}(P)$

Demonstration: Demonstration of Graham scan algorithm is shown in Fig. 5.4 for a small point set. Note that the points are labeled in lexicographically ascending order of their coordinates (with horizontal x-axis and vertical y-axis). In (a), the point p_4 is the current (last) vertex in U. Before processing p_4, the current upper hull chain (for p_1, p_2, p_3) was $U = \{p_1, p_3\}$. When p_4 is encountered, it is added to U to get $U = \{p_1, p_3, p_4\}$ (see step 4 of CONVEX-HULL-GRAHAM-SCAN algorithm). Since the turn at p_3 — considering the triplet (p_1, p_3, p_4) — is not a right turn, p_3 is removed from U.

The case shown in (b) is a typical and interesting instance of Graham scan algorithm. Just before processing p_8, the vertices of the upper hull chain up to p_7 were $U = \{p_1, p_4, p_5, p_6, p_7\}$. When p_8 is considered, the turn at vertex p_7 with p_6 as the preceding vertex and p_8 as the following vertex fails to be a right turn, whereby p_7 is removed from U; After removal of p_7 from U, (p_5, p_6, p_8) becomes the current triplet of vertices which again fails as a right turn, whereby p_6 is removed. The next check is for (p_4, p_5, p_8) which is again not a right turn and so p_5 is removed; and finally, due to right turn for (p_1, p_4, p_8), only these three vertices (in order) remain in U.

5.2.4 A gift-wrapping algorithm (Jarvis march)

This algorithm is based on observation 5.5. But here, the approach is analogous to wrapping a flexible (and sufficiently long) thread around a set of pins (the gift) fitted on a wooden board (the point set, P). If we fix one end of the thread on the leftmost pin (the point, p'_1), hold it vertically up (along $(+)y$-axis), and go on wrapping the thread over the pins in clockwise direction, the thread will hit (and turn around) only those pins that represent the vertices of the convex hull of P.

To start with, therefore, we find the point p'_1 with minimum x-coordinate in P (in case of multiple minima, we consider p'_1 as the point with minimum x and minimum y), and initialize the vertex set of $\mathcal{C}(P)$ with p'_1. We compute the angle θ (w.r.t. $(+)y$-axis) and the Euclidean distance d for each other point of P. The next hull vertex would be the point with minimum θ, and in case of multiple points with minimum θ, we consider among them the point with maximum d. We repeat this process for the latest hull vertex obtained so far, until we get the vertex, say p'_n, with maximum x-coordinate at which the angle θ of each other point of P exceeds 180^0. The vertex p'_n is the rightmost point (the bottommost among the rightmost points, if there are multiple points in P with maximum x-coordinate) in P, and hence the rightmost vertex (the second rightmost vertex between the two vertices, if there are multiple points in P with maximum x-coordinate) of $\mathcal{C}(P)$. Thus we get the upper hull chain,

We repeat the above process from p'_n onwards to find the subsequent lower hull vertices with θ measured in clockwise direction w.r.t. $(-)y$-axis, and finally we reach the start vertex p'_1 to obtain the entire $\mathcal{C}(P)$.

Algorithm 5.3 CONVEX-HULL-JARVIS-MARCH(P)

01. find the point $p'_1 \in P$ with minimum x-coordinate (or the bottommost such point in case of a tie)
02. find the point $p'_n \in P$ with maximum x-coordinate (or the topmost such point in case of a tie)
03. $\mathcal{C}(P) \leftarrow p'_1$
04. $p \leftarrow p'_1$
05. **while** $p \neq p'_n$ ▷ upper hull chain
06. find the point q that has the smallest polar angle (θ)
    with respect to p measured w.r.t. $+y$-axis as shown in Fig. 5.5
    (or the furthest such point to p in case of a tie)
07. $\mathcal{C}(P) \leftarrow \mathcal{C}(P) \cup \{q\}$
08. $p \leftarrow q$
09. **while** $p \neq p'_1$ ▷ lower hull chain
10. find the point q that has the smallest polar angle (θ)
    with respect to p measured w.r.t. $-y$-axis as shown in Fig. 5.5
    (or the furthest such point to p in case of a tie)
11. $\mathcal{C}(P) \leftarrow \mathcal{C}(P) \cup \{q\}$
12. $p \leftarrow q$

Time complexity: Finding out the minimum x-coordinate needs $O(n)$ time. Computing θ and d of the other points of P w.r.t. the current hull vertex needs $O(n)$ time. So, if there are h

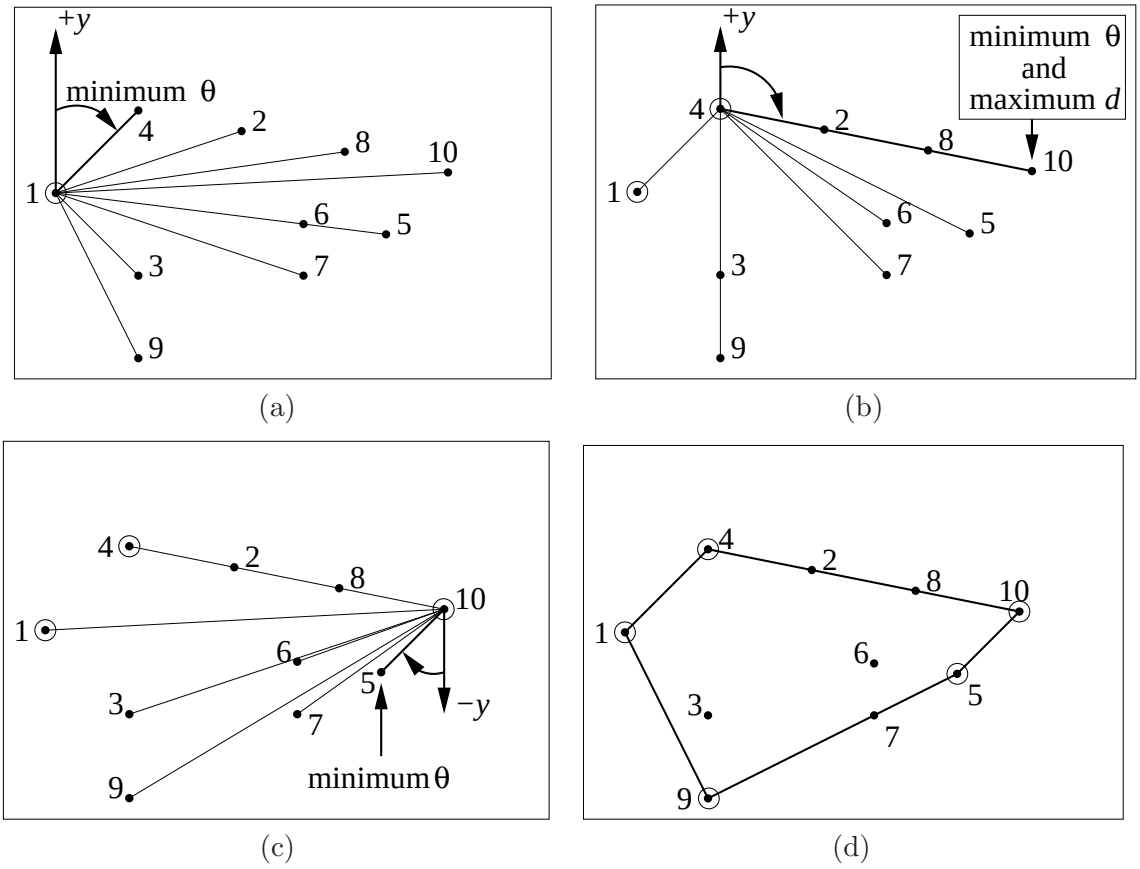


Figure 5.5: Demonstration of Jarvis march algorithm on a small point set, the points being labeled by their indices for simplicity. The encircled point in (c) represent the vertices of $\mathcal{C}(P)$. Note that, here the points are not lexicographically sorted as in Graham scan algorithm (Fig. 5.4); only the point with minimum x-coordinate and that with maximum x-coordinate have been labeled by 1 and $(n =)10$ respectively, and the other points are just arbitrarily labeled.

hull vertices in the output, then the total time complexity is given by $O(nh)$, which is attractive when the convex hull has a relatively very small complexity (i.e., $\mathcal{C}(P)$ contains a very small number of vertices) compared to the number of points in P. Note: Such an algorithm whose time complexity depends not only on the input size (n) but also on the output size (h) is said to be an ***output-sensitive algorithm***.

5.2.5 A Divide-and-Conquer Algorithm

1. If $|P| \leq 3$, then compute $\mathcal{C}(P)$ in $O(1)$ time and return.
2. Partition P into P_L and P_R using the median of x-coordinates of points in P.
3. Recursively compute $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$ for P_L and P_R respectively.
4. Merge $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$ to get $\mathcal{C}(P)$ by computing their lower and upper tangents and deleting all the vertices of $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$ (except the vertex points of tangency) lying between these two tangents[1](Fig. 5.6).

[1] A ***lower (upper) tangent*** is the straight line that touches the two hulls such that all the vertices of the

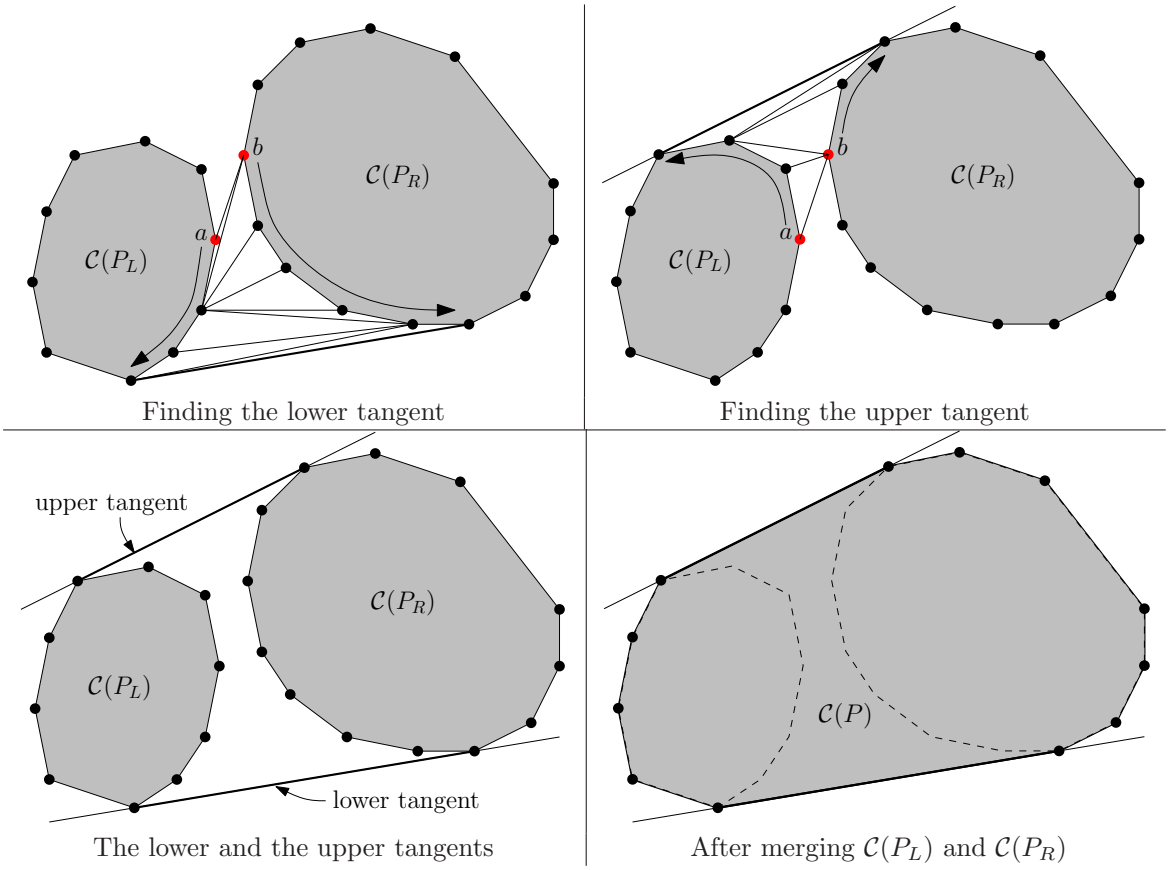


Figure 5.6: Divide-and-conquer approach to find the convex hull $\mathcal{C}(P)$ of P: Recursively finding $\mathcal{C}(P)$ using the convex hulls $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$ corresponding to P_L and P_R, and their lower and upper tangents.

Finding the Lower Tangent:
Let a be the rightmost vertex of $\mathcal{C}(P_L)$ and b be the leftmost vertex of $\mathcal{C}(P_R)$.

(a) while ab is not a lower tangent for $\mathcal{C}(P_L)$ and $\mathcal{C}(P_R)$

i. while ab is not a lower tangent[1] to $\mathcal{C}(P_L)$
$a \leftarrow a + 1$ (move a clockwise)

ii. while ab is not a lower tangent to $\mathcal{C}(P_R)$
$b \leftarrow b - 1$ (move b counterclockwise)

(b) return ab

The upper tangent can be obtained in a similar manner.

Time complexity: While finding a tangent, each vertex on each hull will be checked at most once (e.g., $a + 1$ will be checked only once—whether it lies to the left of the directed ray $\overrightarrow{ba}$). Hence, the runtime to find the two tangents is $O(|\mathcal{C}(P_L)| + |\mathcal{C}(P_R)|) = O(n)$.

two hulls lie above (below) the tangent.

[1] ab is not a lower tangent if the vertex $a+1$ of $\mathcal{C}(P_L)$, lying next to a, lies to the left of the ray/vector directed from b to a.

Hence, Step 4 (merging) of the algorithm needs $O(n)$ time. Step 1 needs $O(1)$ time and Step 2 needs $O(n)$ time for median-based partitioning. Step 3 is recursive. Thus, $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \log n)$.