なお、挿入ソートでは、挿入位置を決定するために、Step 1 で最大 1 回、Step 2 で最大 2 回、$\cdots$、Step $n-1$ で最大 $n-1$ 回の比較が必要となるので、計算量は

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

となります。更に、オーダ記法で表すと $O(n^2)$ となります。

挿入ソートもデータ列によって計算量が変動するアルゴリズムです。最も良いデータ列は、挿入位置を決定するのにどの Step でも比較回数が 1 回で済む場合 (非常に稀なケース) で、その計算量は $n-1$ となります。一般には、挿入位置が平均的に (Step i において等確率 $\frac{i}{2}$ 回の比較で) 見つかると仮定して、計算量

$$\frac{1}{2}+\frac{2}{2}+\cdots+\frac{n-1}{2}=\frac{\left(\frac{1}{2}+\frac{n-1}{2}\right)\cdot(n-1)}{2}=\frac{n^2-n}{4}$$

を求めると、やはりオーダは $O(n^2)$ となります。

問題 1 「挿入ソートによるソーティングプログラム」を作成しなさい。

問題 2 1, 2, $\cdots$, 10 で構成されるデータ列の内、挿入ソートを用いると計算量が最大となるデータ列と最小となるデータ列をそれぞれ求めなさい。

9.5 クイックソート

クイックソートは、データ列から適切な基準値を選び[5]、その基準値より小さな要素で構成されたデータ列と大きな要素で構成されたデータ列に分割することを繰り返し実行することでソーティングを行ないます (分割は、分割されたデータ列の要素の個数が 1 個になるまで再帰的に行なう)。アルゴリズムは、

Step 1 $\{a_1,\ a_2,\ \cdots,\ a_n\}$ から適切な基準値を選ぶ (ここでは a_1 を選ぶ)。基準値より小さな要素 $(a'_i < a_1,\ i=1,\cdots,j)$ で構成されたデータ列 $\{a'_1,\ a'_2,\ \cdots,\ a'_j\}$ と大きな要素 $(a''_i \geq a_1,\ i=1,\cdots,k)$ で構成されたデータ列 $\{a''_1,\ a''_2,\ \cdots,\ a''_k\}$ に分割する $(j+k=n)$。
* 基準値と**最大で**n 回の比較が必要となる (図 9.2 参照)。

Step 2 $\{a'_1,\ a'_2,\ \cdots,\ a'_j\}$ から基準値 a'_1 を選び、Step 1 と同様に分割する。
$\{a''_1,\ a''_2,\ \cdots,\ a''_k\}$ から基準値 a''_1 を選び、Step 1 と同様に分割する。
* 基準値と**最大で**$n-1$ 回の比較が必要となる (図 9.2 参照)。

$\vdots$ $\vdots$

Step m 分割された各データ列の要素が全て 1 個になれば終了。
* 基準値と**最大で**2 回の比較が必要となる (図 9.2 参照)。

注意：分割されたデータ列の先頭要素を基準値に選んでいます。

注意：Step 数は、再帰的方法による深さを表し、m は最大で $n-1$ となります (図 9.2 参照)。

[5]分割するデータ列の長さを丁度半分にするような基準値の選び方が、基準値の良い選び方です。

となります。従って、データ列 {4, 10, 5, 2, 1, 7, 8, 6, 3, 9} をクイックソートによってソーティングすると表 9.5 のようになります (下記の「クイックソートによるソーティングプログラム」でソーティングした場合)。

START	{4, 10, 5, 2, 1, 7, 8, 6, 3, 9}	
Step 1	{{4, 10, 5, 2, 1, 7, 8, 6, 3, 9}}	4 を基準値に選び、
	{{3, 1, 2, 5, 10, 7, 8, 6, 4, 9}}	並び替え分割する。
Step 2	{{3, 1, 2}, {5, 10, 7, 8, 6, 4, 9}}	3, 5 を基準値に選び、
	{{2, 1, 3}, {4, 10, 7, 8, 6, 5, 9}}	並び替え分割する。
Step 3	{{2, 1}, 3, 4, {10, 7, 8, 6, 5, 9}}	以下、同様の手順を繰り返す。
	{{1, 2}, 3, 4, {9, 7, 8, 6, 5, 10}}	
Step 4	{1, 2, 3, 4, {9, 7, 8, 6, 5}, 10}	
	{1, 2, 3, 4, {5, 7, 8, 6, 9}, 10}	
Step 5	{1, 2, 3, 4, {5, 7, 8, 6}, 9, 10}	
	{1, 2, 3, 4, {5, 7, 8, 6}, 9, 10}	
Step 6	{1, 2, 3, 4, 5, {7, 8, 6}, 9, 10}	
	{1, 2, 3, 4, 5, {6, 8, 7}, 9, 10}	
Step 7	{1, 2, 3, 4, 5, 6, {8, 7}, 9, 10}	
	{1, 2, 3, 4, 5, 6, {7, 8}, 9, 10}	
END	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}	

表 9.5: クイックソートによるソーティングの例

また、Step 1 において、基準値 4 ($= a_1$) より小さな要素で構成されたデータ列と大きな要素で構成されたデータ列に並べ替えを行なう過程を表 9.6 に挙げておきます。

	i								j	
{4,	10,	5,	2,	1,	7,	8,	6,	3,	9}	
{3,	10,	5,	2,	1,	7,	8,	6,	4,	9}	交換する。
	i			j						
{3,	10,	5,	2,	1,	7,	8,	6,	4,	9}	
{3,	1,	5,	2,	10,	7,	8,	6,	4,	9}	交換する。
		i	j							
{3,	1,	5,	2,	10,	7,	8,	6,	4,	9}	
{3,	1,	2,	5,	10,	7,	8,	6,	4,	9}	交換する。
		j	i							
{3,	1,	2 ,	5,	10,	7,	8,	6,	4,	9}	
{3,	1,	2},	{5,	10,	7,	8,	6,	4,	9}	分割する。

表 9.6: Step 1 における並べ替えの過程

注意：基準値がデータ列の最小値の場合は、基準値とそれ以外の要素に分割されます (表 9.5 の Step 5)。

● クイックソートによるソーティングプログラム quicksort.c

```
 1: #include <stdio.h>
 2:
 3: #define N 10
 4:
 5: void quicksort(int a[], int start, int end);
 6: void printarray(int a[], int start, int end);
 7:
 8: int main(void)
 9: {
10:     int a[N+1] = {0, 4, 10, 5, 2, 1, 7, 8, 6, 3, 9};
11:
12:     quicksort(a, 1, N);
13:
14:     return 0;
15: }
16:
17: void quicksort(int a[], int start, int end)
18: {
19:     int i, j, x, tmp;
20:
21:     i = start;
22:     j = end;
23:     x = a[start];
24:
25:     printarray(a, start, end);
26:     while (1) {
27:         while (a[i] < x) i++;
28:         while (a[j] > x) j--;
29:         if (i >= j) break;
30:         tmp = a[i];
31:         a[i] = a[j];
32:         a[j] = tmp;
33:         i++; j--;
34:         printarray(a, start, end);
35;     }
36:     printf("¥n");
37:     if (start < i - 1) quicksort(a, start, i - 1);
38:     if (j + 1 < end) quicksort(a, j + 1, end);
39: }
```

```
40:
41: void printarray(int a[], int start, int end)
42: {
43:     int i;
44:
45:     for (i = 1; i <= N; i++) {
46:         if (i == start) printf("{");
47:         printf("%3d ", a[i]);
48:         if (i == end) printf("}");
49:     }
50:     printf("¥n");
51: }
```

注意：関数「quicksort()」は再帰的に呼び出されるため、表 9.5 のような Step 順ではソーティングされません。

なお、クイックソートでは、基準値と Step 1 で最大 n 回、Step 2 で最大 $n-1$ 回、$\cdots$、Step $m = n-1$ (図 9.1 のように分割される場合) で最大 2 回の比較が必要となるので、計算量は

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

となります。更に、オーダ記法で表すと $O(n^2)$ となります。

しかしながら、適切な基準値を取れば図 9.1 のようになることは稀で、Step 回数 m は n に比べて非常に小さくなります。図 9.2 は比較回数が最小の場合で、$n = 2^m$ ($\Longleftrightarrow m = \log n \cdot (\log 2)^{-1}$) が成り立ちます。各 Step の比較回数は n ですから、その計算量は

$$n \cdot m = (\log 2)^{-1} \cdot n \log n$$

となり、オーダ表記すると $O(n \log n)$ となります。

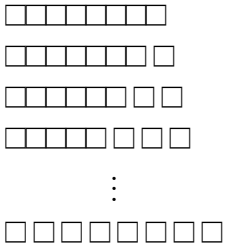

図 9.1: 比較回数が最大となるデータ列

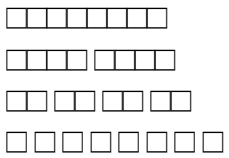

図 9.2: 比較回数が最小となるデータ列

問題 1　「クイックソートによるソーティングプログラム (quicksort.c)」を検証しなさい。

9.6 ヒープソート

ヒープソートは、図 9.3 のような**2 分木** (binary tree) というデータ構造を用い、**ヒープ化** (**ヒープ** (heap) については下記参照) と**ダウンヒープ** (down heap) という 2 つの段階を踏んでソーティングを行ないます。なお、2 分木は「○」で表された**節点** (node) と「—」で表された**枝** (branch) で構成されます。特に、節点の最上位を**根** (root)、子を持たない節点を**葉** (leaf) と呼び、根からある段までの節点を**深さ**と呼びます。また、図 9.4 のように分岐元となる節点を**親** (parent)、分岐先の節点を**子** (child) と呼び、親の要素を a_i とすると 2 つの子の要素は a_{2i} と a_{2i+1} となる親子関係を持っています。

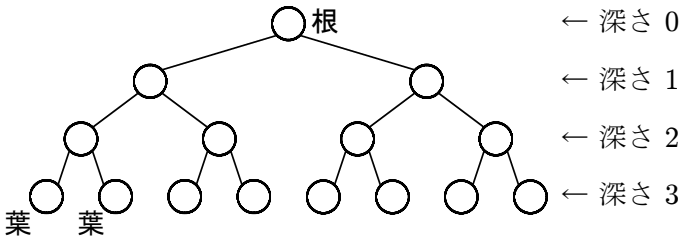

図 9.3: 2 分木

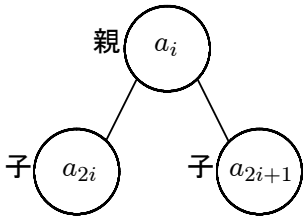

図 9.4: 2 分木の構造

従って、データ列 {4, 10, 5, 2, 1, 7, 8, 6, 3, 9} を 2 分木に対応させると図 9.5 のようになります。

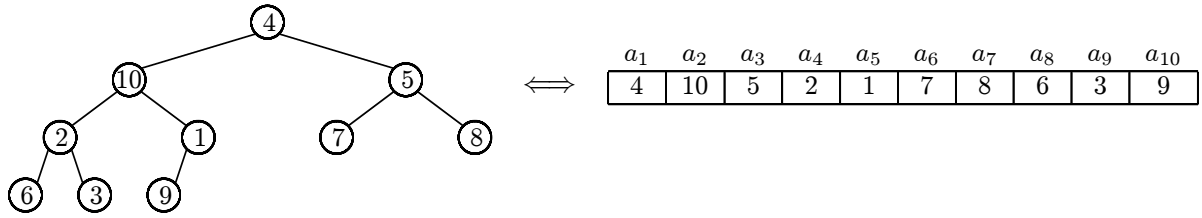

図 9.5: 2 分木と配列の対応

ヒープとダウンヒープは、基本的には同じ作業ですが、ヒープを適用する範囲が異なります。

ヒープ　「ある節点 a_i (親) をヒープする」とは、親 a_i と子 a_{2i}, a_{2i+1} の関係が「$a_i > a_{2i}$」かつ「$a_i > a_{2i+1}$」を満たすように親と子を入れ替え、この操作を下位に向かって (子, 孫, ひ孫, ··· の) 順に繰り返し行なうことです。なお、全ての親の節点で前記の親子関係が成り立つようにヒープを繰り返し適用することを**ヒープ化**と呼びます。

* 親を決定するために 2 回の比較が必要となります。

ダウンヒープ　a_1 と a_i を入れ替え ($a_1 \leftarrow a_i$ かつ $a_i \leftarrow a_1$)、節点 a_1 から a_{i-1} を対象に節点 a_1 をヒープします (節点 a_i から a_n まではソーティング済みとなるため、節点 a_1 から a_{i-1} からなる 2 分木としてヒープを行ないます)。これにより、節点 a_i から a_n までのソーティングが完了します。

* 次の Step では節点 a_1 から a_{i-1} を対象に、ダウンヒープを行ないます。

以上より、ヒープソートのアルゴリズムは、

ヒープ化：

Step 1　節点 $a_{[n/2]}$ をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1) (本当は 2×1 回)。

Step 2　節点 $a_{[n/2]-1}$ をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

⋮　⋮

Step $[n/2]$　節点 a_1 をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

ダウンヒープ：

Step 1　節点 a_n と a_1 を交換し、節点 a_1 と a_{n-1} を対象に節点 a_1 をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

Step 2　節点 a_{n-1} と a_1 を交換し、節点 a_1 と a_{n-2} を対象に節点 a_1 をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

⋮　⋮

Step $n-1$　節点 a_2 と a_1 を交換する。最後は節点 a_1 をヒープする必要がない。
なぜなら、ヒープの対象となる子が存在しないからである。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

(*1)　計算が複雑になるので大きめに見積もります (計算量のオーダ表記の性質より)。

注意：節点 $a_{[n/2]}$ は子を持つ最後の親です ($[n/2]$ 以降の節点は子を持たない)。
注意：2 分木の深さを m (正整数) とします ($m = [(\log 2)^{-1} \cdot \log n - 1] + 1 = [(\log 2)^{-1} \cdot \log n]$)。

$$\because \quad n \fallingdotseq 1 + 2 + \cdots + 2^m = \frac{1 - 2^{m+1}}{1 - 2} = 2^{m+1} - 1 \fallingdotseq 2^{m+1} \quad \Longrightarrow \quad m \fallingdotseq (\log 2)^{-1} \cdot \log n - 1$$

* $[x]$ は x を超えない最大整数を表す (ガウス記号)。

となります。従って、データ列 {4, 10, 5, 2, 1, 7, 8, 6, 3, 9} をヒープソートすると図 9.6 の初期状態からヒープ化 (図 9.7〜図 9.11)、及び、ダウンヒープ (図 9.12〜図 9.17) の過程を経てソーティングが完了します (下記の「ヒープソートによるソーティングプログラム」でソーティングした場合)。

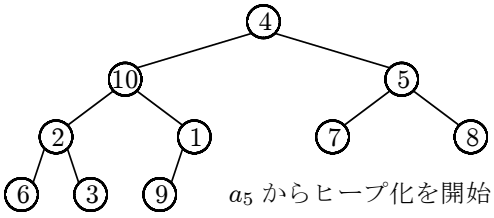

図 9.6: 初期状態

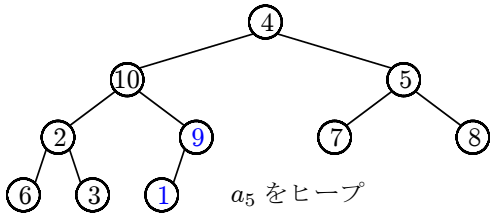

図 9.7: ヒープ化 (Step 1)

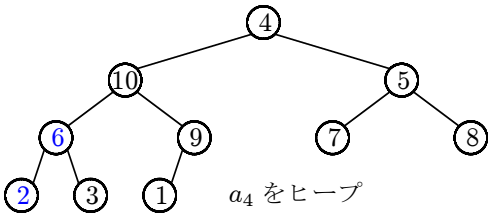

図 9.8: ヒープ化 (Step 2)

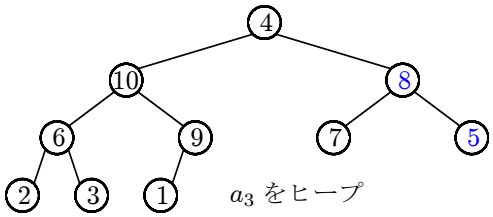

図 9.9: ヒープ化 (Step 3)

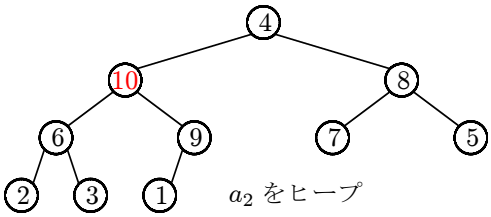

図 9.10: ヒープ化 (Step 4)

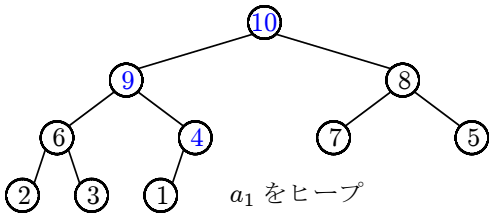

図 9.11: ヒープ化 (Step 5)

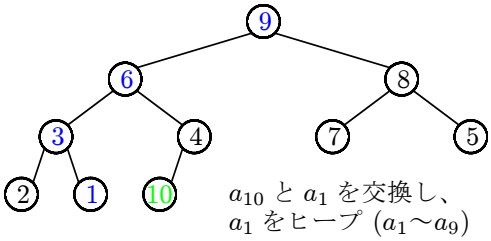

図 9.12: ダウンヒープ (Step 1)

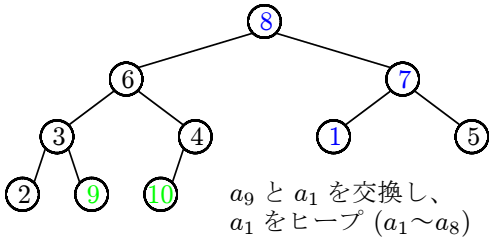

図 9.13: ダウンヒープ (Step 2)

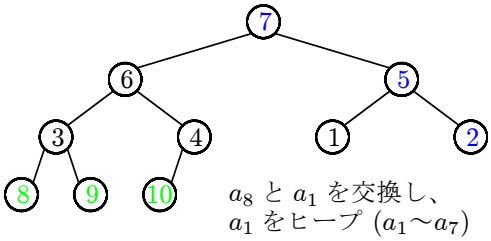

図 9.14: ダウンヒープ (Step 3)

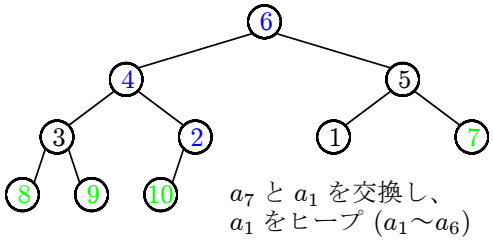

図 9.15: ダウンヒープ (Step 4)

··· 中略 ···

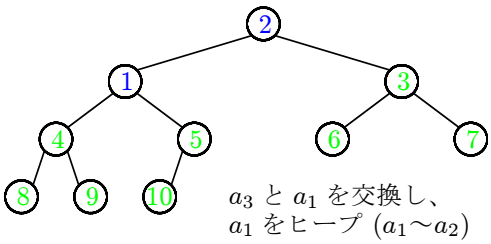

図 9.16: ダウンヒープ (Step 8)

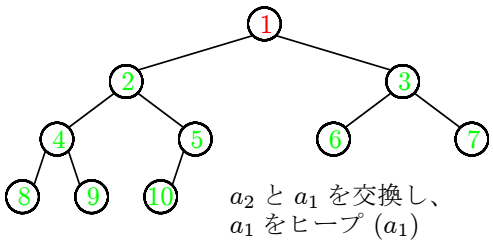

図 9.17: ダウンヒープ (Step 9)

● ヒープソートによるソーティングプログラム　heapsort.c

```
 1: #include <stdio.h>
 2:
 3: #define N 10
 4:
 5: void heapsort(int a[]);
 6: void printarray(int a[]);
 7:
 8: int main(void)
 9: {
10:     int a[N+1] = {0, 4, 10, 5, 2, 1, 7, 8, 6, 3, 9};
11:
12:     heapsort(a);
13:
14:     return 0;
15: }
16:
17: void heapsort(int a[])
18: {
19:     int i, j, k, n = N, x;
20:
21:     for (k = n/2; k >= 1; k--) {
22:         i = k;
23:         x = a[i];
24:         j = 2 * i;
25:         while (j <= n) {
26:             if (j < n && a[j] < a[j+1]) j++;
27:             if (x >= a[j]) break;
28:             a[i] = a[j];
29:             i = j;
30:             j= 2 * i;
31:         }
32:         a[i] = x;
33:         printarray(a);
34:     }
35:     printf("¥n");
36:
```

```
37:     while (n > 1) {
38:         x = a[n];
39:         a[n] = a[1];
40:         n--;
41:         i = 1;
42:         j = 2 * i;
43:         while (j <= n) {
44:             if (j < n && a[j] < a[j+1]) j++;
45:             if (x >= a[j]) break;
46:             a[i] = a[j];
47:             i = j;
48:             j = 2 * i;
49:         }
50:         a[i] = x;
51:         printarray(a);
52:     }
53: }
54:
55: void printarray(int a[])
56: {
57:     int i;
58:
59:     for (i = 1; i <= N; i++) printf("%3d ", a[i]);
60:     printf("¥n");
61: }
```

なお、ヒープソートでは、ヒープ化の各 Step で最大 $2 \cdot m$ $(= 2(\log 2)^{-1} \log n)$ 回、ダウンヒープの各 Step で最大 $2 \cdot m$ $(= 2(\log 2)^{-1} \log n)$ 回の比較が必要となるので、計算量は

$$\left(\frac{n}{2}\right) \cdot (2(\log 2)^{-1} \log n) + (n-1) \cdot (2(\log 2)^{-1} \log n) = 3(\log 2)^{-1} n \log n - 2(\log 2)^{-1} \log n$$

となります。更に、オーダ記法で表すと $O(n \log n)$ となります。ヒープソートの計算量のオーダはクイックソートと同じですが、平均的な計算速度はクイックソートに劣ります。逆に、クイックソートに比べデータ列の良し悪しに左右されないという長所を持ちます。

問題 1 「ヒープソートによるソーティングプログラム (heapsort.c)」を検証しなさい。

最後に、これまで紹介したソーティングアルゴリズムの計算量のオーダ表記について、表 9.7 にまとめておきます (**重要なのは平均的な場合**)。

	最良の場合	最悪の場合	平均的な場合
選択ソート	$O(n^2)$	$O(n^2)$	$O(n^2)$
バブルソート	$O(n)$	$O(n^2)$	$O(n^2)$
挿入ソート	$O(n)$	$O(n^2)$	$O(n^2)$
クイックソート	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
ヒープソート	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

表 9.7: ソーティングアルゴリズムの計算量の比較