今日の論文

QUICKSORT WITHOUT A STACK

Branislav Ďurian
VÚVT Žilina, k.ú.o, Nerudová 33, 01001 Žilina, Czechoslovakia

ABSTRACT: The standard Quicksort algorithm requires a stack of size $O(\log_2 n)$ to sort a set of n elements. We introduce a simple nonrecursive version of Quicksort, which requires only a constant, $O(1)$ additional space because the unsorted subsets are searched instead of stacking their boundaries as in the standard Quicksort. Our $O(1)$-space Quicksort is probably the most efficient of all the sorting algorithms which need a constant workspace only.

KEYWORDS: Algorithm, O(1)-space, Quicksort, Searching, Sorting, Stack.

- Branislav Ďurian, *Mathematical Foundations of Computer Science (MFCS)*, 1986
- `https://link.springer.com/chapter/10.1007/BFb0016252`

そもそも Quicksort ってどんなだっけ？

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []: return [] # 空のリストはソート済み
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

そもそも Quicksort ってどんなだっけ？

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []: return [] # 空のリストはソート済み
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

今回は整数リストを昇順に $\cdots \leq \cdots \leq \cdots$ ソートすることにします。

そもそも Quicksort ってどんなだっけ？

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []: return [] # 空のリストはソート済み
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

今回は整数リストを昇順に $\cdots \le \cdots \le \cdots$ ソートすることにします。
例. `qsort([3, 2, 1, 4, 3, 5])`

そもそも Quicksort ってどんなだっけ？

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []: return [] # 空のリストはソート済み
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

今回は整数リストを昇順に $\cdots \leq \cdots \leq \cdots$ ソートすることにします。
例. `qsort([3, 2, 1, 4, 3, 5])` なら `pivot` $= 3$ の `Rest` $= [2, 1, 4, 3, 5]$

そもそも Quicksort ってどんなだっけ？

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []: return [] # 空のリストはソート済み
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

今回は整数リストを昇順に $\cdots \leq \cdots \leq \cdots$ ソートすることにします。
例. `qsort([3, 2, 1, 4, 3, 5])` なら `pivot` $= 3$ の `Rest` $=$ [`2, 1, 4, 3, 5`]
　3 未満を集めて `Left` $=$ `[2, 1]`, 3 以上を集めて `Right` $=$ `[4, 3, 5]`

そもそも Quicksort ってどんなだっけ？

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []: return [] # 空のリストはソート済み
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

今回は整数リストを昇順に $\cdots \le \cdots \le \cdots$ ソートすることにします。

例. `qsort([3, 2, 1, 4, 3, 5])` なら `pivot` $= 3$ の `Rest` $=$ [2, 1, 4, 3, 5]

　3 未満を集めて `Left` $=$ `[2, 1]`, 3 以上を集めて `Right` $=$ `[4, 3, 5]`

　部分リストを個別撃破; 格好良く言うと「帰納法の仮定から」以下が成立:

　　`qsort(Left) = [1, 2]`, `qsort(Right) = [3, 4, 5]`

そもそも Quicksort ってどんなだっけ？

手抜きクイックソートを Python で書くと

```python
def qsort(L):
  match L:
    case []: return [] # 空のリストはソート済み
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

今回は整数リストを昇順に $\cdots \leq \cdots \leq \cdots$ ソートすることにします。

例. `qsort([3, 2, 1, 4, 3, 5])` なら `pivot` $= 3$ の `Rest` $= [2, 1, 4, 3, 5]$

3 未満を集めて `Left` $=$ `[2, 1]`, 3 以上を集めて `Right` $=$ `[4, 3, 5]`

部分リストを個別撃破; 格好良く言うと「帰納法の仮定から」以下が成立:

`qsort(Left) = [1, 2]`, `qsort(Right) = [3, 4, 5]`

あとは `pivot` を挟めば完成: $[1, 2, \underline{3}_{\texttt{pivot}}, 3, 4, 5]$

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []:              # 空のリストはソート済み
      return []
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

このクイックソートの 実行時間**ではなくて** 必要メモリに注目します。

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []:              # 空のリストはソート済み
      return []
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

このクイックソートの 実行時間**ではなくて** 必要メモリに注目します。

- Left とか Right という補助リストを作るためのメモリ
- 一つの qsort が、二つの再帰呼び出し qsort(Left), qsort(Right) をするので関数のコールスタックのためのメモリ

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []:              # 空のリストはソート済み
      return []
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

このクイックソートの 実行時間**ではなくて** 必要メモリに注目します。

- Left とか Right という補助リストを作るためのメモリ これ削れます; 例えば、教科書的な両側から交換して中央に向かうやつ。また後でやります。
- 一つの qsort が、二つの再帰呼び出し qsort(Left), qsort(Right) をするので関数のコールスタックのためのメモリ

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []:               # 空のリストはソート済み
      return []
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot 未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot 以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

このクイックソートの 実行時間**ではなくて** 必要メモリに注目します。

- Left とか Right という補助リストを作るためのメモリ ⬅ これ削れます; 例えば、教科書的な両側から交換して中央に向かうやつ。また後でやります。
- 一つの qsort が、二つの再帰呼び出し qsort(Left), qsort(Right) をするので関数のコールスタックのためのメモリ ⬅ これを削るのが本題

手抜きクイックソートを Python で書くと

```
def qsort(L):
  match L:
    case []:              # 空のリストはソート済み
      return []
    case pivot, *Rest: # 先頭要素 pivot と、残り Rest に分解
      Left  = [ x for x in Rest if x <  pivot ] # pivot未満を集める
      Right = [ y for y in Rest if y >= pivot ] # pivot以上を集める
      return qsort(Left) + [pivot] + qsort(Right) # 再帰呼び出し
```

このクイックソートの 実行時間**ではなくて** 必要メモリに注目します。

- Left とか Right という補助リストを作るためのメモリ ⬅ これ削れます; 例えば、教科書的な両側から交換して中央に向かうやつ。また後でやります。
- 一つの qsort が、二つの再帰呼び出し qsort(Left), qsort(Right) をするので関数のコールスタックのためのメモリ ⬅ これを削るのが本題

我々は $O(1)$ の extra space しか許されていないので、コールスタックはダメ禁

ソートアルゴリズムの計算量比較

https://en.wikipedia.org/wiki/Sorting_algorithm より

アルゴリズム名	平均時間計算量	最悪時間計算量	空間計算量	stable?
Quicksort	$O(n \log n)$	$O(n^2)$	$\underline{O(\log n)}$	No
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Block sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes

ソートアルゴリズムの計算量比較

https://en.wikipedia.org/wiki/Sorting_algorithm より

アルゴリズム名	平均時間計算量	最悪時間計算量	空間計算量	stable?
Quicksort	$O(n\log n)$	$O(n^2)$	$\underline{O(\log n)}$	No
Heap sort	$O(n\log n)$	$O(n\log n)$	$O(1)$	No
Merge sort	$O(n\log n)$	$O(n\log n)$	$O(n)$	Yes
Block sort	$O(n\log n)$	$O(n\log n)$	$O(1)$	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes

Theorem

スタックなし *Quick sort* で「$O(n\log n)$, $O(n^2)$, $\underline{O(1)}$, *No*」を達成できる。

ソートアルゴリズムの計算量比較

https://en.wikipedia.org/wiki/Sorting_algorithm より

アルゴリズム名	平均時間計算量	最悪時間計算量	空間計算量	stable?
Quicksort	$O(n \log n)$	$O(n^2)$	$\underline{O(\log n)}$	No
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Block sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Yes
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes

Theorem

スタックなし *Quick sort* で「$O(n \log n)$, $O(n^2)$, $\underline{O(1)}$, *No*」を達成できる。

ところで、Block sort はなにもの……？ https://en.wikipedia.org/wiki/Block_sort

```
procedure IQS(var n: integer);
 {the O(1)-space version of QUICKSORT to sort elements in
  A[1..n], sentinels: A[n+1]:=maxval-1, A[n+2]:=maxval,  maxval=∞
  Another improvements from [2] can be applied, too.}
var i,j,l,r,m: integer;
var p: integer;
begin
  l:=1;r:=n+1;m:=9;
  repeat begin
    while r-l>m do begin
      i:=l;j:=r;
      p:=A[l];  {or choosing the pivot as the median of 3 elements}
      repeat
        repeat i:=i+1; until A[i]>=p;
        repeat j:=j-1; until A[j]<=p;
        if i<j then swap(A[i],A[j]);
      until i>=j;
      A[l]:=A[j];A[j]:=p;
      swap(A[i],A[r]);   {instead of pushing on the stack}
      r:=j
    end;
    l:=r;repeat l:=l+1; until A[l]<>p;
    if l<=n then begin  {instead of popping from the stack
                         sequential search for bounds follows}
      p:=A[l];r:=l;
      repeat r:=r+1; until A[r]>p;
      r:=r-1;A[l]:=A[r];A[r]:=p
    end;
  end;
  until l>n;
  insertsort(n)
end.
```

⬅ 論文はこれで出来ると主張している

なので、こっからはこれ（を勝手に単純化したもの）を理解するために話を進める

```
     procedure IQS(var n: integer);
      {the O(1)-space version of QUICKSORT to sort elements in
       A[1..n], sentinels: A[n+1]:=maxval-1, A[n+2]:=maxval, maxval=∞
       Another improvements from [2] can be applied, too.}
     var i,j,l,r,m: integer;
     var p: integer;
     begin
1      l:=1;r:=n+1;m:=9;
       repeat begin
2        while r-l>m do begin
3          i:=l;j:=r;
4          p:=A[l];  {or choosing the pivot as the median of 3 elements}
           repeat
5            repeat i:=i+1; until A[i]>=p;
6            repeat j:=j-1; until A[j]<=p;
7            if i<j then swap(A[i],A[j]);
8          until i>=j;
9          A[l]:=A[j];A[j]:=p;
10         swap(A[i],A[r]);   {instead of pushing on the stack}
11         r:=j
         end;
12       l:=r;repeat l:=l+1; until A[l]<>p;
13       if l<=n then begin  {instead of popping from the stack
                              sequential search for bounds follows}
14         p:=A[l];r:=l;
15         repeat r:=r+1; until A[r]>p;
16         r:=r-1;A[l]:=A[r];A[r]:=p
         end;
       end;
17     until l>n;
18     insertsort(n)
     end.
```

⬅ 論文はこれで出来ると主張している

なので、こっからはこれ（を勝手に単純化したもの）を理解するために話を進める

いやいや……
そもそも、スタック消せたらなんやねん?

```
procedure IQS(var n: integer);
 {the O(1)-space version of QUICKSORT to sort elements in
  A[1..n], sentinels: A[n+1]:=maxval-1, A[n+2]:=maxval,  maxval=∞
  Another improvements from [2] can be applied, too.}
var i,j,l,r,m: integer;
var p: integer;
begin
  l:=1;r:=n+1;m:=9;
  repeat begin
    while r-l>m do begin
      i:=l;j:=r;
      p:=A[l];  {or choosing the pivot as the median of 3 elements}
      repeat
        repeat i:=i+1; until A[i]>=p;
        repeat j:=j-1; until A[j]<=p;
        if i<j then swap(A[i],A[j]);
      until i>=j;
      A[l]:=A[j];A[j]:=p;
      swap(A[i],A[r]);   {instead of pushing on the stack}
      r:=j
    end;
    l:=r;repeat l:=l+1; until A[l]<>p;
    if l<=n then begin  {instead of popping from the stack
                         sequential search for bounds follows}
      p:=A[l];r:=l;
      repeat r:=r+1; until A[r]>p;
      r:=r-1;A[l]:=A[r];A[r]:=p
    end;
  end;
  until l>n;
  insertsort(n)
end.
```

⬅ 論文はこれで出来ると主張している

なので、こっからはこれ（を勝手に単純化したもの）を理解するために話を進める

いやいや……
そもそも、スタック消せたらなんやねん?

一般的な回答🙋：速度を（そんな）落とさずに使用メモリ削れたら嬉しいやん！
（逆に、使用メモリちょっと増やして爆速になると嬉しいことは実用上多い）

```
procedure IQS(var n: integer);
 {the O(1)-space version of QUICKSORT to sort elements in
  A[1..n], sentinels: A[n+1]:=maxval-1, A[n+2]:=maxval, maxval=∞
  Another improvements from [2] can be applied, too.}
var i,j,l,r,m: integer;
var p: integer;
begin
  l:=1;r:=n+1;m:=9;
  repeat begin
    while r-l>m do begin
      i:=l;j:=r;
      p:=A[l]; {or choosing the pivot as the median of 3 elements}
      repeat
        repeat i:=i+1; until A[i]>=p;
        repeat j:=j-1; until A[j]<=p;
        if i<j then swap(A[i],A[j]);
      until i>=j;
      A[l]:=A[j];A[j]:=p;
      swap(A[i],A[r]); {instead of pushing on the stack}
      r:=j
    end;
    l:=r;repeat l:=l+1; until A[l]<>p;
    if l<=n then begin {instead of popping from the stack
                        sequential search for bounds follows}
      p:=A[l];r:=l;
      repeat r:=r+1; until A[r]>p;
      r:=r-1;A[l]:=A[r];A[r]:=p
    end;
  end;
  until l>n;
  insertsort(n)
end.
```

⬅️ 論文はこれで出来ると主張している

なので、こっからはこれ（を勝手に単純化したもの）を理解するために話を進める

いやいや……
そもそも、スタック消せたらなんやねん?

一般的な回答🙋: 速度を（そんな）落とさずに使用メモリ削れたら嬉しいやん！（逆に、使用メモリちょっと増やして爆速になると嬉しいことは実用上多い）

真の答え🔪💊🪓: 学生が再帰呼び出しを全く理解してくれない。それなら再帰呼び出ししない版のクイックソートを実装しましょうか、というヤケクソスピリット🔥

最初の一歩: 古典的な配列の要素入れ替え法で、`Left` と `Right` を削減する

```
def qsort(L):
    ...
    Left  = [ x for x in Rest if x <  pivot ] # ← これ消す
    Right = [ y for y in Rest if y >= pivot ] # ← これも消す
    ...
```

最初の一歩: 古典的な配列の要素入れ替え法で、Left と Right を削減する

```
def qsort(L):
    ...
    Left  = [ x for x in Rest if x <  pivot ] # ← これ消す
    Right = [ y for y in Rest if y >= pivot ] # ← これも消す
    ...
```

アイディア【左側に 値 $<$ pivot を集める、右側に 値 $\geq$ pivot を集める】
（つまり、フツーにクイックソート実装する時にやるやつです）

最初の一歩: 古典的な配列の要素入れ替え法で、Left と Right を削減する

```
def qsort(L):
    ...
    Left  = [ x for x in Rest if x <  pivot ] # ← これ消す
    Right = [ y for y in Rest if y >= pivot ] # ← これも消す
    ...
```

アイディア【左側に 値 < pivot を集める、右側に 値 ≥ pivot を集める】
（つまり、フツーにクイックソート実装する時にやるやつです）

- インデックス変数 i と j を準備する。
- i は、左から右に向かって、 L[i] < pivot の間は突き進む (i += 1)
- j は、右から左に向かって、 L[j] ≥ pivot の間は突き進む (j -= 1)

最初の一歩: 古典的な配列の要素入れ替え法で、Left と Right を削減する

```
def qsort(L):
    ...
    Left  = [ x for x in Rest if x <  pivot ] # ← これ消す
    Right = [ y for y in Rest if y >= pivot ] # ← これも消す
    ...
```

アイディア【左側に 値 $<$ pivot を集める、右側に 値 $\geq$ pivot を集める】
（つまり、フツーにクイックソート実装する時にやるやつです）

- インデックス変数 i と j を準備する。
- i は、左から右に向かって、 L[i] $<$ pivot の間は突き進む (i += 1)
- j は、右から左に向かって、 L[j] $\geq$ pivot の間は突き進む (j -= 1)
- L[i] $\geq$ pivot かつ L[j] $<$ pivot で止まったら

最初の一歩: 古典的な配列の要素入れ替え法で、Left と Right を削減する

```
def qsort(L):
    ...
    Left  = [ x for x in Rest if x <  pivot ] # ← これ消す
    Right = [ y for y in Rest if y >= pivot ] # ← これも消す
    ...
```

アイディア【左側に 値 $<$ pivot を集める、右側に 値 $\geq$ pivot を集める】
（つまり、フツーにクイックソート実装する時にやるやつです）

- インデックス変数 i と j を準備する。
- i は、左から右に向かって、 $\mathtt{L}[\mathtt{i}] < \mathtt{pivot}$ の間は突き進む (i += 1)
- j は、右から左に向かって、 $\mathtt{L}[\mathtt{j}] \geq \mathtt{pivot}$ の間は突き進む (j -= 1)
- $\mathtt{L}[\mathtt{i}] \geq \mathtt{pivot}$ かつ $\mathtt{L}[\mathtt{j}] < \mathtt{pivot}$ で止まったら
- $\mathtt{L}[\mathtt{i}], \mathtt{L}[\mathtt{j}]$ の値を「入れ替え」る:

$$\mathtt{L}[\mathtt{i}] \geq \mathtt{pivot} \wedge \mathtt{L}[\mathtt{j}] < \mathtt{pivot} \quad \overset{\text{値を入れ替え}}{\Longrightarrow} \quad \mathtt{L}[\mathtt{i}] < \mathtt{pivot} \wedge \mathtt{L}[\mathtt{j}] \geq \mathtt{pivot}$$

最初の一歩: 古典的な配列の要素入れ替え法で、Left と Right を削減する

```
def qsort(L):
  ...
  Left  = [ x for x in Rest if x <  pivot ] # ← これ消す
  Right = [ y for y in Rest if y >= pivot ] # ← これも消す
  ...
```

アイディア【左側に 値 $<$ pivot を集める、右側に 値 $\geq$ pivot を集める】
（つまり、フツーにクイックソート実装する時にやるやつです）

- インデックス変数 i と j を準備する。
- i は、左から右に向かって、 $\mathtt{L}[\mathtt{i}] < \mathtt{pivot}$ の間は突き進む (i += 1)
- j は、右から左に向かって、 $\mathtt{L}[\mathtt{j}] \geq \mathtt{pivot}$ の間は突き進む (j -= 1)
- $\mathtt{L}[\mathtt{i}] \geq \mathtt{pivot}$ かつ $\mathtt{L}[\mathtt{j}] < \mathtt{pivot}$ で止まったら
- $\mathtt{L}[\mathtt{i}], \mathtt{L}[\mathtt{j}]$ の値を「入れ替え」る:

$$\mathtt{L}[\mathtt{i}] \geq \mathtt{pivot} \wedge \mathtt{L}[\mathtt{j}] < \mathtt{pivot} \quad \overset{\text{値を入れ替え}}{\Longrightarrow} \quad \mathtt{L}[\mathtt{i}] < \mathtt{pivot} \wedge \mathtt{L}[\mathtt{j}] \geq \mathtt{pivot}$$

Q. なんでリストでランダムアクセスしてるねん
A. Python のリストは dynamic array (C++でいう vector）でした

Partitioning の計算例

$\mathtt{L} = [4^{\mathtt{p}}, 3, 4, 9, 1, 5, 5, 2, 8, 7]$ (pivot = 4) の場合

start	$[\,4^p, 3^i, 4, 9, 1, 5, 5, 2, 8, 7^j\,]$	($L[\,i\,] < p$ の間は i++、$L[\,j\,] \geq p$ の間は j−−)
$\Rightarrow$	$[\,4^p, 3, 4^i, 9, 1, 5, 5, 2^j, 8, 7\,]$	($L[\,i\,] \geq p, L[\,j\,] < p$ を値交換で解決)
$\Rightarrow$	$[\,4^p, 3, 2^i, 9, 1, 5, 5, 4^j, 8, 7\,]$	(進めるだけ進むを再開)
$\Rightarrow$	$[\,4^p, 3, 2, 9^i, 1^j, 5, 5, 4, 8, 7\,]$	(また交換する)
$\Rightarrow$	$[\,4^p, 3, 2, 1^i, 9^j, 5, 5, 4, 8, 7\,]$	(進めるだけ進むを再開)
$\Rightarrow$	$[\,4^p, 3, 2, 1^j, 9^i, 5, 5, 4, 8, 7\,]$	STOP!! $j \leq i$ になったので
$\Rightarrow$	$[\,1^p, 3, 2, 4^j, 9^i, 5, 5, 4, 8, 7\,]$	pivot と j を入れ替えて仕上げ

Partitioning の計算例

$\texttt{L} = [4^{\texttt{p}}, 3, 4, 9, 1, 5, 5, 2, 8, 7]$ (pivot $= 4$) の場合

$$\begin{array}{lll}
\text{start} & [\,4^p, 3^i, 4, 9, 1, 5, 5, 2, 8, 7^j\,] & (L[\,i\,] < p \text{ の間は } i{+}{+}\text{、} L[\,j\,] \geq p \text{ の間は } j{-}{-}) \\
\Rightarrow & [\,4^p, 3, 4^i, 9, 1, 5, 5, 2^j, 8, 7\,] & (L[\,i\,] \geq p, L[\,j\,] < p \text{ を値交換で解決}) \\
\Rightarrow & [\,4^p, 3, 2^i, 9, 1, 5, 5, 4^j, 8, 7\,] & (\text{進めるだけ進むを再開}) \\
\Rightarrow & [\,4^p, 3, 2, 9^i, 1^j, 5, 5, 4, 8, 7\,] & (\text{また交換する}) \\
\Rightarrow & [\,4^p, 3, 2, 1^i, 9^j, 5, 5, 4, 8, 7\,] & (\text{進めるだけ進むを再開}) \\
\Rightarrow & [\,4^p, 3, 2, 1^j, 9^i, 5, 5, 4, 8, 7\,] & \text{STOP!! } j \leq i \text{ になったので} \\
\Rightarrow & [\,1^p, 3, 2, 4^j, 9^i, 5, 5, 4, 8, 7\,] & \text{pivot と } j \text{ を入れ替えて仕上げ}
\end{array}$$

ひたすら交換を繰り返し、i と j がすれ違った ($j \leq i$) 後で
pivot と位置 j の値を入れ替えるところまでやると、次のことが成立する：

- 右側には `pivot` 以上の値だけ: $\forall k \geq j.\ L[k] \geq \texttt{pivot}$
- 左側には `pivot` 未満の値だけ: $\forall k < j.\ L[k] < \texttt{pivot}$

Partitioning の計算例

L $= [4^{\mathrm{p}}, 3, 4, 9, 1, 5, 5, 2, 8, 7]$ (pivot = 4) の場合

start	$[\,4^p, 3^i, 4, 9, 1, 5, 5, 2, 8, 7^j\,]$	($L[\,i\,] < p$ の間は $i{+}{+}$、$L[\,j\,] \geq p$ の間は $j{-}{-}$)
$\Rightarrow$	$[\,4^p, 3, 4^i, 9, 1, 5, 5, 2^j, 8, 7\,]$	($L[\,i\,] \geq p, L[\,j\,] < p$ を値交換で解決)
$\Rightarrow$	$[\,4^p, 3, 2^i, 9, 1, 5, 5, 4^j, 8, 7\,]$	(進めるだけ進むを再開)
$\Rightarrow$	$[\,4^p, 3, 2, 9^i, 1^j, 5, 5, 4, 8, 7\,]$	(また交換する)
$\Rightarrow$	$[\,4^p, 3, 2, 1^i, 9^j, 5, 5, 4, 8, 7\,]$	(進めるだけ進むを再開)
$\Rightarrow$	$[\,4^p, 3, 2, 1^j, 9^i, 5, 5, 4, 8, 7\,]$	STOP!! $j \leq i$ になったので
$\Rightarrow$	$[\,1^p, 3, 2, 4^j, 9^i, 5, 5, 4, 8, 7\,]$	pivot と j を入れ替えて仕上げ

ひたすら交換を繰り返し、i と j がすれ違った ($j \leq i$) 後で
pivot と位置 j の値を入れ替えるところまでやると、次のことが成立する：

- 右側には `pivot` 以上の値だけ: $\forall k \geq j.\ L[k] \geq \texttt{pivot}$
- 左側には `pivot` 未満の値だけ: $\forall k < j.\ L[k] < \texttt{pivot}$

よって Left 相当 $L[0..j\,)$ をソートし
Right 相当 $L[\,j..\omega)$ もソートすれば、全体がソートできる。

再帰バージョンの Quicksort

```
def qsort(L, l, r): # L[l..r) = L[l], L[l+1], ..., L[r-1] をソート
  if l == r: return # 空部分リストのソートなので、何もしない

  pivot = L[l]; # 左端を pivot にする
  i = l+1; j = r-1; # 初期位置をセット

  # Partitioning part
  while True:
    while i < j and L[i] <  pivot: i += 1;
    while l < j and L[j] >= pivot: j -= 1;
    if j <= i: break
    else: L[i], L[j] = L[j], L[i] # 値の入れ替え

  L[l] = L[j]; L[j] = pivot;
  qsort(L, l, j)
  qsort(L, j+1, r) # qsort(L, j, r) でも別に良い
```

Call Stack を取り除くための準備

```
def qsort(L, l, r):
  # L[l..r) = L[l], L[l+1], ..., L[r-1] をソート
  # ... 中略 ...
  qsort(L, l, j)
  qsort(L, j+1, r)
```

一つの qsort は内部で qsort を 2 回呼び出す。その様子は木だと描きやすい:

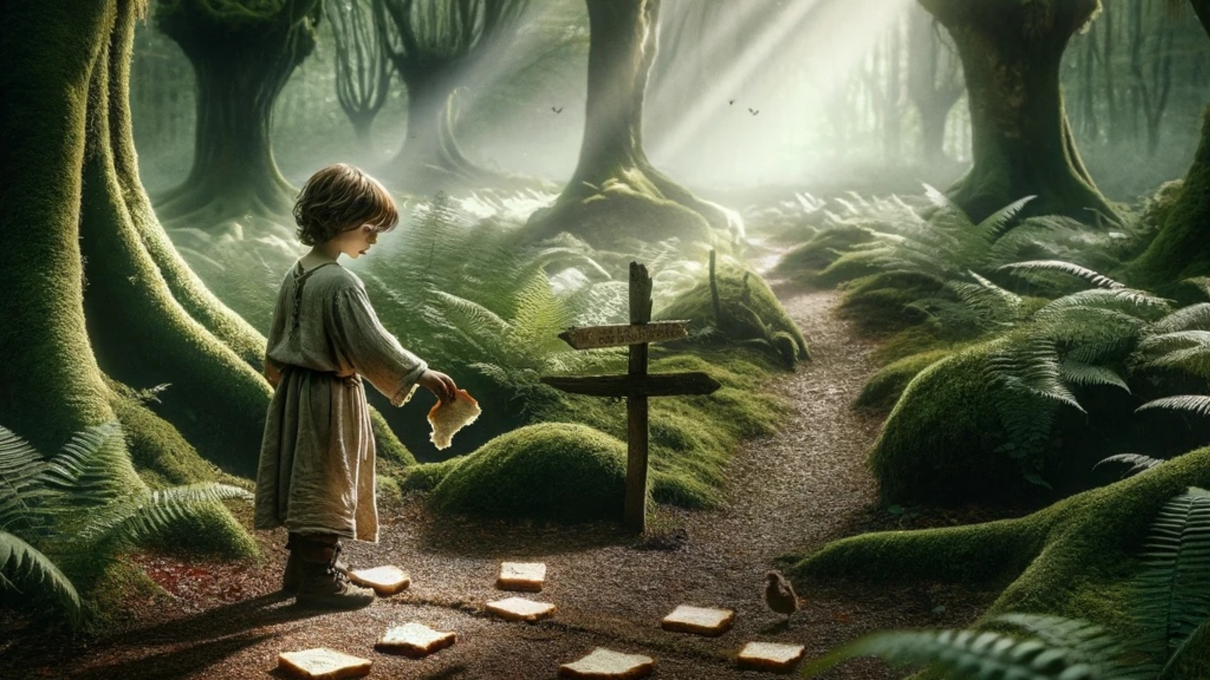
Call Stack を取り除くための準備

```
def qsort(L, l, r):
  # L[l..r) = L[l], L[l+1], ..., L[r-1] をソート
  # ... 中略 ...
  qsort(L, l, j)
  qsort(L, j+1, r)
```

一つの qsort は内部で qsort を 2 回呼び出す。その様子は木だと描きやすい:

$$
\begin{tikzcd}
 & & & Q(l,r) \arrow[dll] \arrow[drr] & & & \\
 & Q(l,j) \arrow[dl] \arrow[dr] & & & & Q(j+1,r) \arrow[dl] \arrow[dr] & \\
Q(l,x) \arrow[d] & & Q(x+1,j) \arrow[d] & & Q(j+1,y) \arrow[d] & & Q(y+1,r) \arrow[d] \\
\vdots & & \vdots & & \vdots & & \vdots
\end{tikzcd}
$$

（見易さのために L は端折って、関数名も Q にしてます）



木を on-the-fly に作って traverse しようとすると、
自分の親ノードを覚えるために stack を使うことになる。

$$
\begin{tikzcd}[column sep=small]
 & & & & {}^{1}Q(l,r) \arrow[dl] \arrow[dr] & \\
 & & & {}^{2}Q(l,j) \arrow[dl] \arrow[dr] & & Q(j+1,r) \\
 & & {}^{3}Q(l,x) \arrow[dl] \arrow[dr] & & Q(x+1,j) & \\
 & {}^{4}Q(l,l) & & {}^{5}Q(l+1,x) \arrow[dl] \arrow[dr] & & \\
 & & {}^{6}Q(\cdots) & & Q(\cdots) &
\end{tikzcd}
$$

↑左部分リストと右部分リストに partition して、
左部分リストを先に計算していっている様子を書いているつもり。

木を on-the-fly に作って traverse しようとすると、
自分の親ノードを覚えるために stack を使うことになる。

$$
\begin{array}{ccccccccc}
 & & & & & & {}^{1}Q(l,r) & & \\
 & & & & & \swarrow & & \searrow & \\
 & & & & {}^{2}Q(l,j) & & & & Q(j+1,r) \\
 & & & \swarrow & & \searrow & & & \\
 & & {}^{3}Q(l,x) & & & & Q(x+1,j) & & \\
 & \swarrow & & \searrow & & & & & \\
{}^{4}Q(l,l) & & & & {}^{5}Q(l+1,x) & & & & \\
 & & & \swarrow & & \searrow & & & \\
 & & {}^{6}Q(\cdots) & & & & Q(\cdots) & &
\end{array}
$$

↑左部分リストと右部分リストに partition して、
左部分リストを先に計算していっている様子を書いているつもり。
（親ノードを覚えて律儀に辿っても良いし、
次に着目すべき部分リストを次々に stack に積んでいく＝継続を形成する のでも良い。）

スタックがないとどう困る？

先の（途中まで展開した）計算木に対応する、次の状況を考える:

$$[[\mathsf{Sorted}\ p_3\ [\underline{\mathsf{Sorting}}\ p_4\ \mathsf{Unsorted}_4]]\ p_2\ \mathsf{Unsorted}_2]\ p_1\ \mathsf{Unsorted}_1$$

いま見ている部分 $\underline{\mathsf{Sorting}}$ の計算が完了したら、
その次は $\mathsf{Unsorted}_4$ に着手したい、という状況。

スタックがないとどう困る？

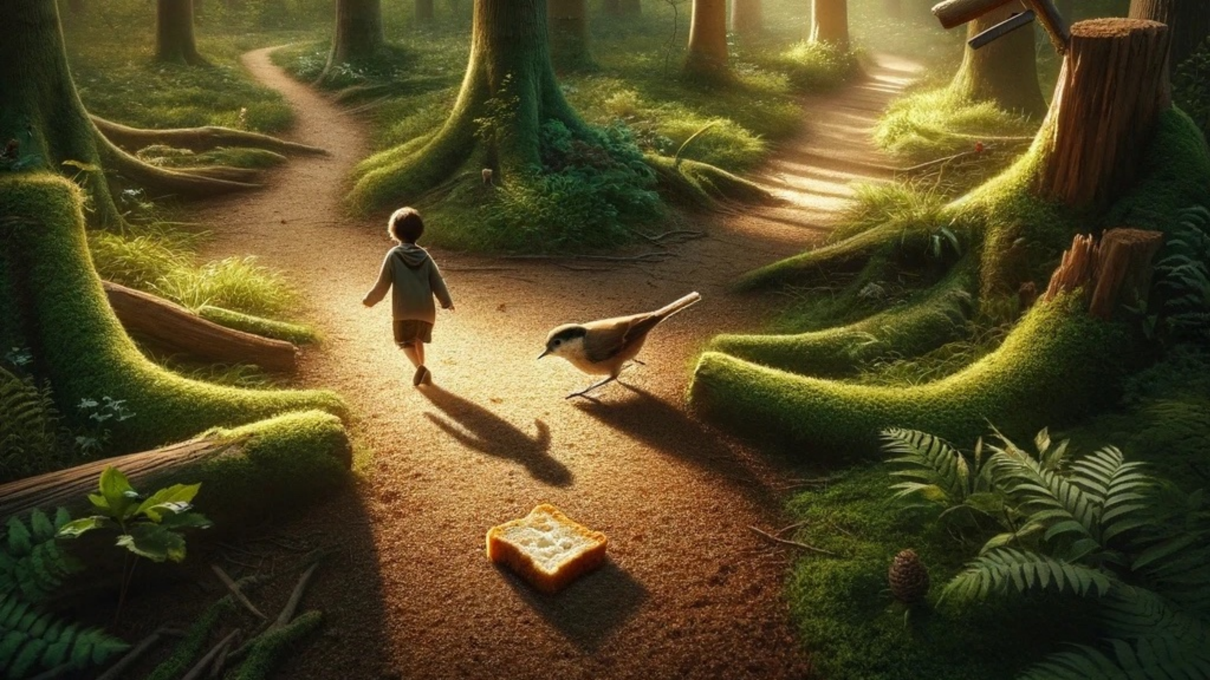
先の（途中まで展開した）計算木に対応する、次の状況を考える:

$$[[\mathsf{Sorted}\ p_3\ [\underline{\mathsf{Sorting}}\ p_4\ \mathsf{Unsorted}_4]]\ p_2\ \mathsf{Unsorted}_2]\ p_1\ \mathsf{Unsorted}_1$$

いま見ている部分 $\underline{\mathsf{Sorting}}$ の計算が完了したら、
その次は $\mathsf{Unsorted}_4$ に着手したい、という状況。

スタックを使って「次にするべき計算」を記憶していれば何の問題もなく達成できる。

しかし、スタックが使えないとなると……



↓のように、次の計算先への目印がなくなる:

$$\text{Sorted } \underline{\text{Sorting}}\ p \text{ Unsorted}$$

↓のように、次の計算先への目印がなくなる:

$$\text{Sorted}\ \underline{\text{Sorting}}\ p\ \text{Unsorted}$$

今見ているところ = <u>Sorting</u> より右側の全て = Unsorted をソート対象にしても
ソート自体はできる。

↓のように、次の計算先への目印がなくなる:

$$\text{Sorted } \underline{\text{Sorting}}\ p \text{ Unsorted}$$

今見ているところ $=$ Sorting より右側の全て $=$ Unsorted をソート対象にしても
ソート自体はできる。が、計算コストとしては
先立って partition した部分が無駄になるので勿体無い。
というより、selection sort っぽくなり、いつでも $O(N^2)$ となりそう。

↓のように、次の計算先への目印がなくなる:

$$\text{Sorted } \underline{\text{Sorting}}\ p \text{ Unsorted}$$

今見ているところ = Sorting より右側の全て = Unsorted をソート対象にしても
ソート自体はできる。が、計算コストとしては
先立って partition した部分が無駄になるので勿体無い。
というより、selection sort っぽくなり、いつでも $O(N^2)$ となりそう。

pivot さえ他と区別できれば良さそうなので、マーク／印 をつければ良い？

$$\text{sorted } \dot{p}\ \underline{\text{sorting}}\ \dot{p} \text{ Unsorted } \dot{p} \text{ Unsorted } \dot{p} \text{ Unsorted}$$

↓のように、次の計算先への目印がなくなる:

$$\text{Sorted}\ \underline{\text{Sorting}}\ p\ \text{Unsorted}$$

今見ているところ = $\underline{\text{Sorting}}$ より右側の全て = Unsorted をソート対象にしても
ソート自体はできる。が、計算コストとしては
先立って partition した部分が無駄になるので勿体無い。
というより、selection sort っぽくなり、いつでも $O(N^2)$ となりそう。

pivot さえ他と区別できれば良さそうなので、マーク／印 をつければ良い？

$$\text{sorted}\ \dot{p}\ \underline{\text{sorting}}\ \dot{p}\ \text{Unsorted}\ \dot{p}\ \text{Unsorted}\ \dot{p}\ \text{Unsorted}$$

こうやって、現在地のすぐ右と、その次の pivot の間を処理すればうまくいく。

例えば **1word** が **64bit** なのに、**63bit** 以下の値しか来ていないなど、**flag** 用の **bit** が使えるなら追加のメモリはない。

が、一般には stack と同程度のメモリが必要になり、 $O(1)$ は無理。

↓のように、次の計算先への目印がなくなる:

$$\text{Sorted}\ \underline{\text{Sorting}}\ p\ \text{Unsorted}$$

今見ているところ = $\underline{\text{Sorting}}$ より右側の全て = Unsorted をソート対象にしても
ソート自体はできる。が、計算コストとしては
先立って partition した部分が無駄になるので勿体無い。
というより、selection sort っぽくなり、いつでも $O(N^2)$ となりそう。

pivot さえ他と区別できれば良さそうなので、マーク／印 をつければ良い？

$$\text{sorted}\ \dot{p}\ \underline{\text{sorting}}\ \dot{p}\ \text{Unsorted}\ \dot{p}\ \text{Unsorted}\ \dot{p}\ \text{Unsorted}$$

こうやって、現在地のすぐ右と、その次の pivot の間を処理すればうまくいく。

例えば 1word が 64bit なのに、63bit 以下の値しか来ていないなど、flag 用の bit が使えるなら追加のメモリはない。

が、一般には stack と同程度のメモリが必要になり、 $O(1)$ は無理。
もう少し賢くできないか？

アイディア

$$\overbrace{\text{Sorted}}^{<p}\ \overbrace{\text{Partitioning}}^{<p}\ p\ \overbrace{\text{Unsorted}}^{\geq p} \xrightarrow{\text{partition 完了 by } p'}$$

$$\text{Sorted}\ \ L\ p'\ r_1\ r_2 \ldots r_n\ p\ \ \text{Unsorted} \xrightarrow{p\ \leftrightarrow\ r_1}$$

$$\text{Sorted}\ \ \underline{L}\ p'\ p\ r_2 \ldots r_n\ r_1\ \ \text{Unsorted}$$

アイディア

$$\overbrace{\text{Sorted}}^{<p}\ \overbrace{\text{Partitioning}}^{<p}\ p\ \overbrace{\text{Unsorted}}^{\geq p} \xrightarrow{\text{partition 完了 by } p'}$$

$$\text{Sorted}\ \ L\ p'\ r_1\ r_2 \ldots r_n\ p\ \ \text{Unsorted} \xrightarrow{p \leftrightarrow r_1}$$

$$\text{Sorted}\ \ \underline{L}\ p'\ p\ r_2 \ldots r_n\ r_1\ \ \text{Unsorted}$$

partition して、左部分リスト L の計算に入る「前」に、
右部分リストの先頭 r_1 を、Partitioning 全体の上界 p と交換する。

アイディア

$$\overbrace{\text{Sorted}}^{<p}\ \overbrace{\text{Partitioning}}^{<p}\ p\ \overbrace{\text{Unsorted}}^{\geq p} \xrightarrow{\text{partition 完了 by } p'}$$

$$\text{Sorted}\ \ L\ p'\ r_1\ r_2 \ldots r_n\ p\ \ \text{Unsorted} \xrightarrow{p \leftrightarrow r_1}$$

$$\text{Sorted}\ \ \underline{L}\ p'\ p\ r_2 \ldots r_n\ r_1\ \ \text{Unsorted}$$

partition して、左部分リスト L の計算に入る「前」に、
右部分リストの先頭 r_1 を、Partitioning 全体の上界 p と交換する。 さて、

$$\to \cdots \to \text{Sorted}\ \ \underline{L_{\text{sorted}}}\ \ p'\ p\ r_2 \ldots r_n\ r_1\ \ \text{Unsorted}$$

計算を進めて↑この状況になったら、いまちょうど計算が終わった部分 $\underline{L_{\text{sorted}}}$ の
二つ右の要素 つまり p に着目する。

アイディア

$$\overbrace{\text{Sorted}}^{<p}\ \overbrace{\text{Partitioning}}^{<p}\ p\ \overbrace{\text{Unsorted}}^{\geq p} \xrightarrow{\text{partition 完了 by } p'}$$

$$\text{Sorted}\ \ L\ p'\ r_1\ r_2 \ldots r_n\ p\ \ \text{Unsorted} \xrightarrow{p \leftrightarrow r_1}$$

$$\text{Sorted}\ \ \underline{L}\ p'\ p\ r_2 \ldots r_n\ r_1\ \ \text{Unsorted}$$

partition して、左部分リスト L の計算に入る「前」に、
右部分リストの先頭 r_1 を、Partitioning 全体の上界 p と交換する。 さて、

$$\to \cdots \to \text{Sorted}\ \ \underline{L_{\text{sorted}}}\ \ p'\ p\ r_2 \ldots r_n\ r_1\ \ \text{Unsorted}$$

計算を進めて↑この状況になったら、いまちょうど計算が終わった部分 $\underline{L_{\text{sorted}}}$ の
二つ右の要素 つまり p に着目する。

この p から右向きに走査し、p 未満 $(< p)$ の値が出ている間は進む。

$$\text{Sorted}\ \ \underline{L_{\text{sorted}}}\ p'\ p\ \overbrace{r_2 \ldots r_n\ \check{r}_1}^{<p}\ \ \overbrace{\text{Unsorted}}^{\geq p}$$

アイディア

$$\overbrace{\text{Sorted}}^{<p}\ \overbrace{\text{Partitioning}}^{<p}\ p\ \overbrace{\text{Unsorted}}^{\geq p} \xrightarrow{\text{partition 完了 by } p'}$$

$$\text{Sorted}\ \ L\ p'\ r_1\ r_2 \ldots r_n\ p\ \ \text{Unsorted} \xrightarrow{p \leftrightarrow r_1}$$

$$\text{Sorted}\ \ \underline{L}\ p'\ p\ r_2 \ldots r_n\ r_1\ \ \text{Unsorted}$$

partition して、左部分リスト L の計算に入る「前」に、
右部分リストの先頭 r_1 を、Partitioning 全体の上界 p と交換する。 さて、

$$\to \cdots \to \text{Sorted}\ \ \underline{L_{\text{sorted}}}\ \ p'\ p\ r_2 \ldots r_n\ r_1\ \ \text{Unsorted}$$

計算を進めて↑この状況になったら、いまちょうど計算が終わった部分 $\underline{L_{\text{sorted}}}$ の
二つ右の要素 つまり p に着目する。

この p から右向きに走査し、p 未満 $(< p)$ の値が出ている間は進む。

$$\text{Sorted}\ \ \underline{L_{\text{sorted}}}\ p'\ p\ \overbrace{r_2 \ldots r_n\ \check{r}_1}^{<p}\ \ \overbrace{\text{Unsorted}}^{\geq p}$$

【命題: 止まった位置 $\check{\bullet}$ に r_1 がいる。】

アイディア

$$\overbrace{\text{Sorted}}^{<p}\ \overbrace{\text{Partitioning}}^{<p}\ p\ \overbrace{\text{Unsorted}}^{\geq p} \xrightarrow{\text{partition 完了 by } p'}$$

$$\text{Sorted}\ \ L\ p'\ r_1\ r_2 \dots r_n\ p\ \ \text{Unsorted} \xrightarrow{p \leftrightarrow r_1}$$

$$\text{Sorted}\ \ \underline{L}\ p'\ p\ r_2 \dots r_n\ r_1\ \ \text{Unsorted}$$

partition して、左部分リスト L の計算に入る「前」に、
右部分リストの先頭 r_1 を、Partitioning 全体の上界 p と交換する。 さて、

$$\to \cdots \to \text{Sorted}\ \ \underline{L_{\text{sorted}}}\ \ p'\ p\ r_2 \dots r_n\ r_1\ \ \text{Unsorted}$$

計算を進めて↑この状況になったら、いまちょうど計算が終わった部分 $\underline{L_{\text{sorted}}}$ の
二つ右の要素 つまり p に着目する。

この p から右向きに走査し、p 未満 $(< p)$ の値が出ている間は進む。

$$\text{Sorted}\ \ \underline{L_{\text{sorted}}}\ p'\ p\ \overbrace{r_2 \dots r_n\ \check{r}_1}^{<p}\ \ \overbrace{\text{Unsorted}}^{\geq p}$$

【命題: 止まった位置 $\check{\bullet}$ に r_1 がいる。】なので再度交換し、計算を続行できる:

$$\text{Sorted}\ \ L_{\text{sorted}}\ p'\ \underline{r_1\ r_2 \dots r_n}\ p\ \ \text{Unsorted}$$

(※ 右部分リストが空の場合は交換しない。復帰時に p のすぐ右が $\geq p$ なので空だったことは分かる。)

論文のアイディアを、例で確認

リスト $L = [5, 0, 4, 3, 2, 1, 7, 6]$ を考える。

論文のアイディアを、例で確認

リスト $L = [5, 0, 4, 3, 2, 1, 7, 6]$ を考える。末尾に、番兵として無限大 ∞ を加える:

$$[5, 0, 4, 3, 2, 1, 7, 6, \infty]$$

論文のアイディアを、例で確認

リスト $L = [5, 0, 4, 3, 2, 1, 7, 6]$ を考える。末尾に、番兵として無限大 ∞ を加える:

$$[5, 0, 4, 3, 2, 1, 7, 6, \infty]$$

理由: Sorted $\overbrace{\text{Sorting}}^{<q}$ q $\overbrace{\text{Unsorted}}^{\geq q}$ の形を invariant にしたいから。

今の場合は、Sorted = 空リスト, Sorting = L, $q = \infty$, Unsorted = 空リスト

論文のアイディアを、例で確認

リスト $L = [5, 0, 4, 3, 2, 1, 7, 6]$ を考える。末尾に、番兵として無限大 ∞ を加える:

$$[5, 0, 4, 3, 2, 1, 7, 6, \infty]$$

理由: Sorted $\overbrace{\text{Sorting}}^{<q}$ q $\overbrace{\text{Unsorted}}^{\geq q}$ の形を invariant にしたいから。
今の場合は、Sorted = 空リスト, Sorting = L, $q = \infty$, Unsorted = 空リスト

まず 5^p での partition を完了させる。左部分リスト=$[1, 0, 4, 3, 2]$; 右部分リスト=$[7, 6]$:

$$\underline{1, 0, 4, 3, 2, 5^p, 7, 6}, \infty$$

論文のアイディアを、例で確認

リスト $L = [5, 0, 4, 3, 2, 1, 7, 6]$ を考える。末尾に、番兵として無限大 ∞ を加える:

$$[5, 0, 4, 3, 2, 1, 7, 6, \infty]$$

理由: Sorted $\overbrace{\text{Sorting}}^{<q}$ q $\overbrace{\text{Unsorted}}^{\geq q}$ の形を invariant にしたいから。
今の場合は、Sorted = 空リスト, Sorting = L, $q = \infty$, Unsorted = 空リスト

まず 5^p での partition を完了させる。左部分リスト=$[1, 0, 4, 3, 2]$; 右部分リスト=$[7, 6]$:

$$\underline{1, 0, 4, 3, 2, 5^p, 7, 6}, \infty$$

上界 ∞ と 右部分リスト先頭 7 を交換する:

$$1, 0, 4, 3, 2, 5, \infty, 6, 7$$

論文のアイディアを、例で確認

リスト $L = [5, 0, 4, 3, 2, 1, 7, 6]$ を考える。末尾に、番兵として無限大 ∞ を加える:

$$[5, 0, 4, 3, 2, 1, 7, 6, \infty]$$

理由: Sorted $\overbrace{\text{Sorting}}^{<q}$ q $\overbrace{\text{Unsorted}}^{\geq q}$ の形を invariant にしたいから。
今の場合は、Sorted = 空リスト, Sorting = L, $q = \infty$, Unsorted = 空リスト

まず 5^p での partition を完了させる。左部分リスト=$[1, 0, 4, 3, 2]$; 右部分リスト=$[7, 6]$:

$$\underline{1, 0, 4, 3, 2, 5^p, 7, 6}, \infty$$

上界 ∞ と 右部分リスト先頭 7 を交換する:

$$1, 0, 4, 3, 2, 5, \infty, 6, 7$$

左部分リストを 1^p で partition し、左部分リスト=$[0]$、右部分リスト=$[4, 3, 2]$ を得る:

$$\underline{0, 1^p, 4, 3, 2}, 5, \infty, 6, 7$$

上界 5 と 右部分リスト先頭 4 を交換し。左部分リストをソートしていく:

$$\underline{0}, 1, 5, 3, 2, 4, \infty, 6, 7$$

左部分リスト [0] はすぐソート完了する。
次は右部分リスト $[4, 3, 2]$ のソートに入りたいので、復元作業をする。

上界 5 と 右部分リスト先頭 4 を交換し。左部分リストをソートしていく:

$$\underline{0}, 1, 5, 3, 2, 4, \infty, 6, 7$$

左部分リスト $[0]$ はすぐソート完了する。
次は右部分リスト $[4, 3, 2]$ のソートに入りたいので、復元作業をする。

ソート完了したてのブロック $[0]$ の「二つ隣の値」に着目 $\dot{5}$ し、< 5 の間は右に進んでいく:

$$0,\ 1,\ \dot{5},\ 3,\ 2,\ \check{4}, \infty, 6, 7$$

上界 5 と 右部分リスト先頭 4 を交換し。左部分リストをソートしていく:

$$\underline{0}, 1, 5, 3, 2, 4, \infty, 6, 7$$

左部分リスト [0] はすぐソート完了する。
次は右部分リスト [4, 3, 2] のソートに入りたいので、復元作業をする。

ソート完了したてのブロック [0] の「二つ隣の値」に着目 $\dot{5}$ し、< 5 の間は右に進んでいく:

$$0,\ 1,\ \dot{5},\ 3,\ 2,\ \check{4}, \infty, 6, 7$$

これ以上進めないところと交換すると、右部分リストが「復元」されるので、ソートスタート:

$$0,\ 1,\ \underline{4, 3, 2},\ 5, \infty, 6, 7$$

上界 5 と 右部分リスト先頭 4 を交換し。左部分リストをソートしていく:

$$\underline{0}, 1, 5, 3, 2, 4, \infty, 6, 7$$

左部分リスト [0] はすぐソート完了する。
次は右部分リスト [4, 3, 2] のソートに入りたいので、復元作業をする。

ソート完了したてのブロック [0] の「二つ隣の値」に着目 $\dot{5}$ し、< 5 の間は右に進んでいく:

$$0,\ 1,\ \dot{5},\ 3,\ 2,\ \check{4}, \infty, 6, 7$$

これ以上進めないところと交換すると、右部分リストが「復元」されるので、ソートスタート:

$$0,\ 1,\ \underline{4, 3, 2},\ 5, \infty, 6, 7$$

さて、この部分のソートが済んで次のようになったとする

$$\to \cdots \to 0, 1,\ \underline{2, 3, 4}, 5, \infty, 6, 7$$

上界 5 と 右部分リスト先頭 4 を交換し。左部分リストをソートしていく:

$$\underline{0}, 1, 5, 3, 2, 4, \infty, 6, 7$$

左部分リスト $[0]$ はすぐソート完了する。
次は右部分リスト $[4, 3, 2]$ のソートに入りたいので、復元作業をする。

ソート完了したてのブロック $[0]$ の「二つ隣の値」に着目 $\dot{5}$ し、< 5 の間は右に進んでいく:

$$0,\ 1,\ \dot{5},\ 3,\ 2,\ \check{4}, \infty, 6, 7$$

これ以上進めないところと交換すると、右部分リストが「復元」されるので、ソートスタート:

$$0,\ 1,\ \underline{4, 3, 2},\ 5, \infty, 6, 7$$

さて、この部分のソートが済んで次のようになったとする

$$\to \cdots \to 0, 1,\ \underline{2, 3, 4}, 5, \infty, 6, 7$$

いま計算完了したブロックの「二つ隣の値」に着目 $\dot{\infty}$ し、次のソート対象を「復元」する:

$$0, 1,\ 2, 3, 4, 5, \underline{7, 6}, \infty$$

上界 5 と 右部分リスト先頭 4 を交換し。左部分リストをソートしていく:

$$\underline{0}, 1, 5, 3, 2, 4, \infty, 6, 7$$

左部分リスト [0] はすぐソート完了する。
次は右部分リスト $[4, 3, 2]$ のソートに入りたいので、復元作業をする。

ソート完了したてのブロック [0] の「二つ隣の値」に着目 $\dot{5}$ し、< 5 の間は右に進んでいく:

$$0,\ 1,\ \dot{5},\ 3,\ 2,\ \check{4}, \infty, 6, 7$$

これ以上進めないところと交換すると、右部分リストが「復元」されるので、ソートスタート:

$$0,\ 1,\ \underline{4, 3, 2},\ 5, \infty, 6, 7$$

さて、この部分のソートが済んで次のようになったとする

$$\to \cdots \to 0, 1,\ \underline{2, 3, 4}, 5, \infty, 6, 7$$

いま計算完了したブロックの「二つ隣の値」に着目 $\dot{\infty}$ し、次のソート対象を「復元」する:

$$0, 1,\ 2, 3, 4, 5, \underline{7, 6}, \infty$$

もうちょっと計算を続けると、無事にソートが完了 $[0, 1, 2, 3, 4, 5, 6, 7, \infty]$ する。

without_stack.py

```python
import copy
import itertools

def R(L):
    l = 0
    N = len(L)
    r = N - 1

    while True:
        while l < r:
            pivot = L[l] # 左端をpivotにする
            i = l+1
            j = r-1

            # Partitioning part
            while True:
                while i < j and L[i] <  pivot: i += 1
                while l < j and L[j] >= pivot: j -= 1
                if j <= i:
                    break
                else:
                    L[i], L[j] = L[j], L[i] # 値の入れ替え

            L[l] = L[j]
            L[j] = pivot

            if j < r-1: # 右部分リストが存在する
                L[j+1], L[r] = L[r], L[j+1] # 値の交換

            r = j # 左部分リストにとりかかる

        assert(l == r)  # 再帰呼び出しのbase case相当になったので

        if r == N-1:
            return # 回復不要＝計算終了

        p = L[r+1]
        # <pの間右に進むことにする
        count = 0
        for idx in range(r+2, N):
            if L[idx] < p:
                count += 1
            else: break
        if count > 0: # 右部分リストがあったことになるので
            L[r+1+count], L[r+1] = L[r+1], L[r+1+count] # 交換する
        l = r+1
        r = r+1 + count
```

まとめ

Quicksort には、明示的・非明示的にスタックが必要で $O(\log n)$ 空間必要か？
と思いきや、スタックを消し去り、空間計算量を $O(1)$ にできた。

しかも、時間計算量（平均 $O(n \log n)$; 最悪 $O(n^2)$）はそのまま（やや遅くはなる）

まとめ

Quicksort には、明示的・非明示的にスタックが必要で $O(\log n)$ 空間必要か？
と思いきや、スタックを消し去り、空間計算量を $O(1)$ にできた。

しかも、時間計算量（平均 $O(n \log n)$; 最悪 $O(n^2)$）はそのまま（やや遅くはなる）

計算量的には、時間と空間には相関があって

- 例えば、メモリを潤沢に持ってくると実行時間が減る（速くなる）し、
- メモリをケチると実行時間が増える（遅くなる)。

今回はメモリをケチった代わりに「再計算＝右部分リストの復元」 を行い
そのぶん時間はかかるが、まあ上手く行きましたという話になった。

まとめ

Quicksort には、明示的・非明示的にスタックが必要で $O(\log n)$ 空間必要か？
と思いきや、スタックを消し去り、空間計算量を $O(1)$ にできた。

しかも、時間計算量（平均 $O(n \log n)$; 最悪 $O(n^2)$）はそのまま（やや遅くはなる）

計算量的には、時間と空間には相関があって

- 例えば、メモリを潤沢に持ってくると実行時間が減る（速くなる）し、
- メモリをケチると実行時間が増える（遅くなる)。

今回はメモリをケチった代わりに「再計算＝右部分リストの復元」 を行い
そのぶん時間はかかるが、まあ上手く行きましたという話になった。

この時間空間トレードオフの話もできるので、教材としては結構良い気がしている。

まとめ

Quicksort には、明示的・非明示的にスタックが必要で $O(\log n)$ 空間必要か？
と思いきや、スタックを消し去り、空間計算量を $O(1)$ にできた。

しかも、時間計算量（平均 $O(n \log n)$; 最悪 $O(n^2)$）はそのまま（やや遅くはなる）

計算量的には、時間と空間には相関があって

- 例えば、メモリを潤沢に持ってくると実行時間が減る（速くなる）し、
- メモリをケチると実行時間が増える（遅くなる)。

今回はメモリをケチった代わりに「再計算＝右部分リストの復元」 を行い
そのぶん時間はかかるが、まあ上手く行きましたという話になった。

この時間空間トレードオフの話もできるので、教材としては結構良い気がしている。

最強のカリキュラム談義をして反映したいので「これは絶対やるべき」
というトピックがある方がいたら、話しかけてください。