トーナメントによるインプレイスソートのアルゴリズム

加悦 周治

1. 概論

本稿の目的は、筆者の思いついたソートのアルゴリズム自体を説明し、これまでに発表されていない物ではないかと提起する物である。筆者はこれを部門トップソートと命名した。

トーナメント戦をしてチャンピオンを選ぶ処理を最後の順位まで繰り返すというのが、ソートの基本的な方法である。
従来からトーナメント戦をするソートは存在するが、データを記憶する配列以外にトーナメント結果の記憶領域を必要とする。2)3) しかるにこのソートではデータ配列自体の中にトーナメントの結果を記録し、ソートデータ以外のメモリ領域は基本的に必要としない。
再帰的処理である為、O(log N)のスタックを必要とするが、再帰的処理としない改造も可能である。

ソートの分類としては、インプレイスソートである。また安定ソートではない。
ソート方法は、基本は交換ソートになる。ただし、選択ソートとのハイブリッドにした速度改善版が本ソートの最速処理となる。

速度性能は最小時間 O(N logN) 最大時間 O(N^2) 平均時間 O(N^2) になる。
速度改善を目的として様々な改造を行い、本稿の中ではこれらの試みも提示する。
改良した物で最大時間, 平均時間共に O(N logN) になったのではと推測するが、確証は得られていない。
クイックソートよりは遅く、最速ではない事は判っている。

2. 詳細

まず、本ソートに必要な条件を述べる。
・ソートされるデータは添字番号でアクセス位置が決まり、この番号は0から始まる配列に入っている必要がある。この添字数値がアルゴリズムに必要である為である。
配列以外の構造の添字が判らないデータはソートできない。
・ソートされるデータ数が予め判っており、その値と同じあるいは超える最初の 2^N (Nは自然数) の値が判っている事。
・ソートされるデータはランダムアクセスできる必要があるが、外部記憶にありメモリ内に全部を格納できなくとも問題ない。

次にアルゴリズムの説明をする。
データを横に並べ、この上にトーナメント戦の木があると考える。
データを前から2個づつに分け、最下層である1回戦の木がある。前から4個づつに分けた上に、2回戦の木がある。同様に8個づつ, 16個づつ .. 2^N 個づつまで、決勝である N回戦まで木が連なる。
各戦いの内容はデータの比較とし、大小関係が反対ならデータを入れ替える。戦いが終わった後、その2つのデータが整列される。
全1回戦終了後、2つのデータの前側にその2つの中のトップが入る。2回戦として、1回戦のトップ同士(1回戦区間の前側)を戦わせ、勝者を前に入れる。4つの一番前にその4つの中のトップが入る。

N回戦終了後、全体のチャンピオン(ソートで最も前に入っているべきデータ)が一番前に入っている事は明白である。
チャンピオンを覗く残りの中でまた同様に最強を選び、残り部分での先頭に入れる。次点者までのデータがソートされた。
これを繰り返す。全順位の戦いが終了し、残りがなくなった時点でソート完了である。
チャンピオン以降の順位を選んでいく時、再度1回戦から全部繰り返す必要はない。2^k (k = 1, 2, 3 ... N-1) 個に分けた各部門内の先頭位置にその部門のトップのデータが入っているからである。残りのデータをできる限り大きくなるように部門に分け、その各部門の先頭にあるトップの中で戦って決めれば良い。
ここで「入っているから」と述べたが、実際にはこの関係は壊れる場合もある。よって再度この関係を作り直す処理をする必要がある。この処理の詳細は後に述べる。

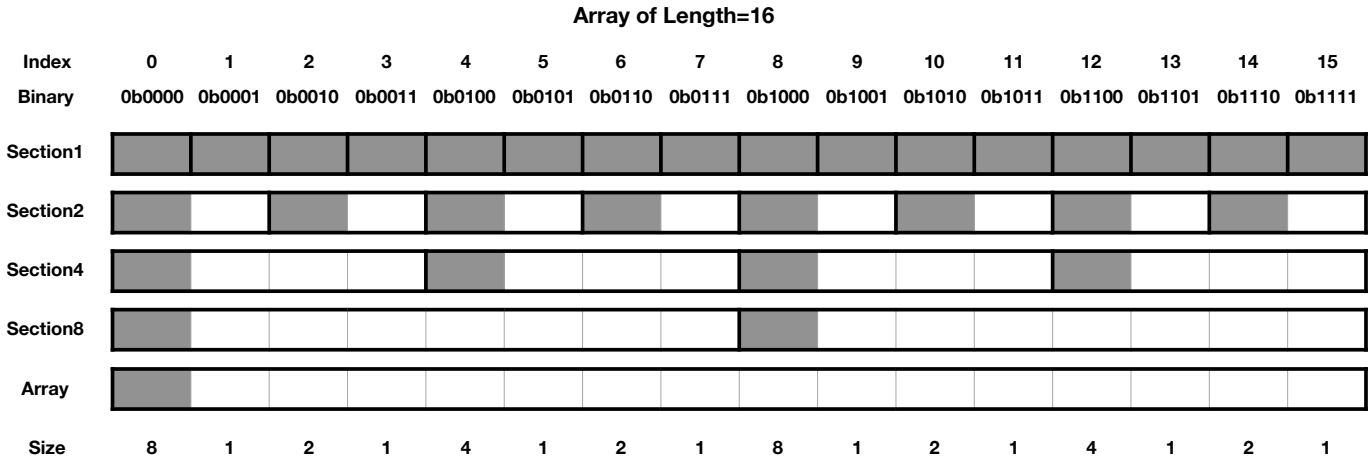
まず、各戦いの処理を下記に示す。(以降示すプログラムリストは全てC言語である。) 入力の superior は前方、inferior は後方の配列の番号であり、この処理内で2つのデータがどちらに入るべきか決まる。

```
void    top_match_for_location(unsigned int superior, unsigned int inferior) {

        if (inferior >= amount)                 // データ数(amount)を超える時、何もしない
                return;
        if (win(inferior, superior)) {          // 比較処理
                exchange(inferior, superior);   // 交換処理
                down_to_the_section(inferior);  // 各部門先頭位置にトップのデータを入れ直す処理
        }
}
```

　データ交換が無ければこの部門に含まれる各部門先頭位置にトップのデータが入る関係は変わらない。
また、交換のあった場合も前方の superior に関しては、それまでのトップを凌ぐデータが来た訳だからこの関係は変わらない事は明らかである。交換があった時の後方の inferior のみ、再度この関係を作り為す処理を呼ぶ必要がある。

　次にデータ位置番号から、その位置を先頭とする最大の部門の大きさ(データ数)を求める方法を述べる。
16個のデータの場合を下図に示す。



2, 4, 8, 16 まで各部門に分けられ、その先頭(色付き位置)にその部門トップが入っている状態を作る。図の配列添字を2進数にしたものから容易にこの大きさが判る。2進数を下位から見て最初に1が出てくる位置の数が、その位置をトップとする最大となる部門の大きさである。例えば 0b???1000 ならば以降 0b1000 (8)個内のトップである。奇数 0x?????1 ならば1個のみでのトップである。

0 の場合は 2^(N-1) になるが、配列添字から計算できず 2^N の値から計算する必要がある。
実際には 0 でこの処理が呼ばれる事は無いが、仮に呼ばれてしまうと無限ループとなるので安全の為対応する処理を入れる。

```
unsigned int    section_size_from_top(unsigned int section_top) {

        unsigned int    section_size;

        if (section_top == 0) {
                return  (maximum_size / 2);             // maximum_size = 2^N (>= amount)
        }
        else {
                section_size = 1;
                while ((section_top & section_size) == 0) {
                        section_size *= 2;
                }
        }
        return  section_size;
}
```

ソート処理自体を下記に示す。上記の処理があれば説明した通りの処理だけでソートが完了する事は判るであろう。0th のチャンピオンが決まる迄と、1th 以降を決めていく2段のループでソートが実現できる。

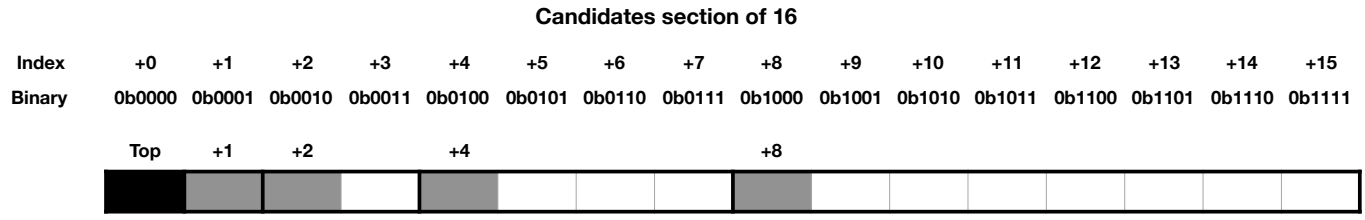
```
void    section_top_sort() {

        unsigned int    size, distance, non_sorted, index;

        for (size = 2, distance = 1; distance < amount; size *= 2, distance *= 2) {
                for (index = 0; index < amount; index += size) {
                        top_match_for_location(index, index + distance);
                }
        }

        for (non_sorted = 1; non_sorted < amount; non_sorted++) {
                for (index = non_sorted; index < amount; ) {
                        index += section_size_from_top(index);
                        top_match_for_location(non_sorted, index);
                }
        }
}
```

最後に各部門先頭位置にトップのデータを入れ直す処理を述べる。16個のデータから成る部門の先頭に前からデータが飛んで来た場合を下図に示す。



次点者以降を決めていく場合と同様に、以降を可能な限り大きい部門に分け、その各トップと戦いを行う訳であるが、位置を計算する処理はもっと簡単化できる。
2分検索木を下から辿るような処理であり、図の通り1から2倍にしていった数を足した所が戦いを行うべき所である。その部門の大きさまで繰り返せば処理は完了である。

```
void    down_to_the_section(unsigned int section_top_location) {

        unsigned int    distance;
        unsigned int    section_size = section_size_from_top(section_top_location);

        for (distance = 1; distance < section_size; distance *= 2) {
                top_match_for_location(section_top_location, section_top_location + distance);
        }
}
```

戦いの処理内部で交換があった時、再度この処理が呼ばれる再帰的処理となっている。
その他一般的な処理はリスト、説明を省略する。ここまででこのソートの全アルゴリズムとなる。

下記にもう少し具体的にした同じ説明を繰り返す。
1. データを前から順番に2個づつの部門に分けていく。部門内で隣同士戦って勝った方を前に、負けた方を後ろにする。
2. データを前から順番に4個づつの部門に分けていく。先の2個づつの試合で勝った方が前にあるので、前側のデータ同士を戦わせ、勝った方を前に、負けた方を後ろにする。4個のなかでトップのデータが先頭になる。
ここで交換が起こらなければ、1. の関係も保たれたままである。交換があった場合、昇格データは 1. の関係も保たれている。降格データは壊れる。よって降格データのみは再度 1. の隣同士戦わせ、 1. の関係を保つようにする。
3. データを前から順番に8個づつの部門に分けていく。先の4個内の試合で最も勝ったデータが前なので、前側のデータ同士を戦わせ、勝った方を前に、負けた方を後ろにする。8個のなかでトップのデータが先頭になる。
ここで交換が起こらなければ、2.1. の関係も保たれたままである。交換があった場合、昇格データは 2.1. の関係も保たれている。降格データは壊れる。よって降格データのみ再度 1.2. の順で試合を行い、 1.2. の関係を保つようにする。2. で交換があれば後ろはまた 1. の試合を行い 1. の関係を保つようにする。
4. データ数を超える 2^N 個づつの部門まで同様の処理を繰り返すと、全データのチャンピオンが先頭に入る。
また 2, 4, 8.. と2^k(k= 1 ... N) の大きさの部門に分けた時、各部門の先頭にその部門のトップが入る形は下の階層まで保たれている。
5. チャンピオンは試合を卒業し、残りの中で再度最強を選ぶ試合を繰り返す。
チャンピオンの後には、1個部門(トップのみ)、2個部門(先頭トップ)、4個部門(先頭トップ)... と並んでいるので、その各トップの中で最強を選べば、次点者が選べる。その位置をトップとする最大の部門の大きさは添字数値の2進数から計算できる。3. までと同様に入れ替えが起こったときは降格データのみは下層から順にその位置にチャレンジする位置にある各部門のトップと戦わせていき、各階層各部門の先頭にトップが来る形を保つ。
6. 同様にしてその位置より後ろにある未確定領域を可能な限り大きな部門になるよう分けていき、そのトップ(最も前)で試合をして勝者を未確定部分先頭に入れる。これを繰り返すと、ソートできる。

引き分けデータがある場合を考える。順番を入れ替える場合に途中のデータを無視して飛ばすので、飛ばしたデータが飛ばした区間内のどれかのデータと引き分けなら引き分けデータの順番は入れ替わる。
つまり、このソートは安定では無い。
引き分けデータは元々入っていた順番に並ぶようにしたい場合は、データと別に同じ大きさのインデックス配列を用意する必要がある。ここに順番を入れておく。交換があった時、この配列も同じ位置を交換する。データ比較で引き分けの時、インデックス配列の数を比較し、元々前にあったデータを勝者とすれば良い。これは安定でない従来他ソートと全く同じである。

3. 特長

まず実行時間について考察する。本ソート処理を比較処理、交換処理、その他のソート処理に分けると、その他のソート処理は他の処理に対して無視できる程に小さい事はプログラムを見れば明らかである。比較処理と交換処理のそれぞれの回数を見る事とする。
のちに示す実際の回数計測値から推測すると、比較回数は最小 (N log2 N)/2 回、最大 (N-1)*N/2回、平均は正確な式は出ないが O(N^2) であろう。
交換回数は最小 0回、最大 (N-1)*N/2回、平均 (N-1)*N/4回であろう。
個数Nの非常に大きい所では、回数の大きい方が効いてくると考えられるので、最小時間 O(N logN) 最大時間 O(N^2) 平均時間 O(N^2) と推察できる。

各部門先頭位置にトップのデータを入れ直す処理が無ければ無いほど、つまり比較でデータの入れ替えが無ければ無いほど、つまり前の方にトップ側のデータがあればあるほど高速に終了する。
これは部門のトップであるデータに勝つデータならば、そのグループの他のデータより大きい事は自明である事を利用して比較が省略できているからである。
最速となるのは最初からソート順に並んでいた場合である。最遅となるのは正反対に並んでいる場合であり、バブルソートと同じ時間を要する事となる。

プログラム自体から明らかなように、ソートデータ自体以外の作業用領域を必要としないが、再帰を含む為、log2 N のスタックを消費する。

その他の本ソートの特長を述べる。
・ソートは前方から決まっていきソート途中でもどこまでソートできたかは明白である。よって全体ソートは必要ない、トップ100までが判れば良い、というような部分ソートには適する。
また全ソートが終わらなくとも、ソートできた所から順番に出力していくといった事が可能になる。
・並列処理との親和性が高い。実際のトーナメント戦が並列に処理される事から明らかである。並列ソートのネットワークと最初の方は非常に似ている。
ソート回数は固定でなくデータにより大きく変化し早く終わる場合もある。よって処理するスレッド数がソートされるデータ数に対して非常に小さい条件では、より適するとも考えられる。
・メモリに全データ読み込みができず、ディスク、ネット上にあるデータをソートする場合のキャッシュとの相性が良いと考えられる。部門のトップデータ入れ替えが生じない限り同じデータが何度も比較に使用される、入れ替え処理でもほとんどが狭い範囲のデータに対する処理が繰り返される事が多い為である。

4. 改良

これまで述べた本ソートの基本版では動作速度が芳しくなく、高速化する事を考える。
5種の改造を試みた。それらについて下記に示す。

4-1. 部門トップソート (摩擦減少変更版)

最も時間のロスになっているのはデータの入れ替えが発生する度に、データが降格して来た部門で部門のトップが先頭に来る関係を作り直さねばならない所である。データ降格が少なくできればできるほど、以降のデータの入れ替えは減らせ処理時間が速くなる。
戦いの度に入れ替えを行わず、勝ったデータを覚えておき最強データとトップの席のデータのみを入れ替えるように修正する。
上記の down_to_the_section を次の処理に入れ替える。

```
void    down_to_the_section_1exchange_only(unsigned int section_top_location) {

        unsigned int            distance;
        unsigned int            really_top, index;
        unsigned int            section_size = section_size_from_top(section_top_location);

        really_top = section_top_location;
        for (distance = 1; distance < section_size; distance *= 2) {
                index = section_top_location + distance;
                if (index >= amount)
                        break;
                if (win(index, really_top))
                        really_top = index;
        }

        if (section_top_location != really_top) {
                exchange(really_top, section_top_location);
                down_to_the_section_1exchange_only(really_top);
        }
}
```

これによりソート方法の種別は交換と選択のハイブリッドに変わる。
チャンピオンより後の各最強データを候補の中から決めていくときも同様の修正ができるが、変更内容自体は上記と同じなのでリストは省略する。

ここで再帰的に down_to_the_section_1exchange_only を呼んでいる所は、C言語なら goto 文を使って書き表す事ができ、再帰によるスタック消費から解放される。
ただし、プログラムリストは構造化プログラミングから外れた汚いプログラムとなるのでここでは示さない。

　動作速度は、後の実測データで示すように最大・平均回数は改善される。最小回数は変わらない。
基本版が1個のデータあたり O(N) 時間がかかりN個のデータで O(N^2) になるのなら、この変更版はで再帰がカットされ1個のデータの処理は O(logN) となり、最大・平均回数は O(N logN) に変わっているのではと推測するが、測定結果から確証は得られなかった。
最大回数となるのは、例えば8個のデータで "5 4 6 3 7 2 1 0", "6 5 4 3 7 2 1 0" となる場合である。筆者には最大時間となる場合の定式化はできなかった。
平均時間も同様である。

これ以降の各変更版は速度改善できたとは言えないので、現状これがこのソートの最良アルゴリズムになる。

4-2. 部門トップソート (全部門交換変更版)

　データが降格して来た部門で部門のトップが先頭に来る関係を保つ別の方法として、先頭データのみでなく部門全体毎入れ替える方法も考えられる。
変更点、リストはあえて説明するほどでないので省略する。
入れ替える部門同士のサイズが合う必要があるので、この変更ができる箇所は限られる。またデータ数が 2^N 個でなければ、2^N 個までどのデータにも負けるデータを詰める必要が有る。データ自体以外の多大なメモリを要する事になり、実用的では無い。
後の測定結果より明らかなように 2^N 個データがある場合と同じになるので、多くの場合速度は遅くなる。
2^N 個同士で比較しても 3-1. より比較回数は減るが交換回数は増える。大きな改善は望めない。
また8個のデータで最大回数となるのは "3 2 7 1 6 5 4 0" であり、最大時間・平均時間の定式化困難なのも同様である。

4-3 ツートップソート

　ここまで部門に分けて「各部門のトップが先頭に来る」規則を定めそれを利用してソートする事を考えた。
ここで規則を増やせばソートの速度は某なるのかというのを考える。各部門の中でトップだけでなく、追加の規則としてナンバー2も決める。このナンバー2は、部門の中間位置に入れる。変更内容、リストは少し複雑になるが本論要旨からずれるので省略する。
プログラムにまだバグがあり測定結果が求まらない場合があるが速度確認用なので現状としている。(ナンバー2が範囲外にくる場合の対応が無い)
後の実測データで示すように比較回数、交換回数共に 3-1. より最小・最大・平均回数は逆に増えてしまう。
またいくつかのデータをランダムに作ってのソートでも比較回数、交換回数共に増えている。
これより規則を増やすに従い、ソート速度は遅くなると考えられる。
また8個のデータで最大回数となるのは "5 4 7 6 1 0 3 2", "5 4 7 6 2 1 3 0", "5 4 7 6 3 2 1 0", "6 5 7 4 1 0 3 2", "6 5 7 4 2 1 3 0", "6 5 7 4 3 2 1 0", "7 6 5 4 1 0 3 2", "7 6 5 4 2 1 3 0", "7 6 5 4 3 2 1 0" であり、最大時間・平均時間の定式化困難なのも同様である。

4-4. 部門トップソート (次セット変更版)

　後半にチャンピオンが含まれるデータを考える。ソートが進み後半部門の先頭にあるチャンピオンと前半部門の先頭とが交換され、チャンピオンが全体の先頭に入る。この時元々前半部門の先頭は後半部門先頭に入り、以降のソートで前半にある各部門トップと共にこの後半先頭のデータは比較される訳であるが、元々後半先頭のデータは前半のトップであり、このデータだけと比較すれば他の前半にある各部門トップとは比較の必要がない。
この例のようにデータ交換でそれ以前のトーナメント記録が失われ、再度同じ比較を行わねばならない事が起こるという事が動作速度を落とす要因である。
これを解決するには別にメモリを取ってこのデータがどのデータに対して勝つ事が確認済みであるか記録しておき、実際に戦う前にそのメモリを確認すれば良い。これには多大なメモリを必要とし、インプレイスなこのソートのメリットを失ってしまう。
ここで最初に述べたケースだけなら、このソートの改良版として対応できる。前半部分のトップを後半トップと交換する時、前半トップのデータを飛ばしてしまわずソート済みエリアの次に置かれるようにすればいいのである。後半先頭には他の関係を壊さない別データを飛ばす。
変更内容、リストは本論要旨からずれるので省略する。
後の実測データで示すように 3-1. より比較回数は改善されるが、交換回数は増える。データ書き込みの必要な交換の方が時間がかかると考えると、速度的には改善されないのではと考えられる。
また8個のデータで最大回数となるのは "7 5 6 4 3 1 2 0" であり、最大時間・平均時間の定式化困難なのも同様である。

4-5. 同時に両方からソート

　2個のデータからなる部門でトップを先頭にセットした時、これは同時にボトムを最後にセットする処理でもある。また4個のデータからなる部門でトップを先頭にセットする時、ボトムのデータを最後にセットする処理は参照データが重ならず、並列に実行できる。以降トップ側の処理と、正反対のボトム側の処理を同時に進めても参照する位置は重ならず、並列に実行できる。
ただし交換発生後には、、トップ、ボトムの関係をセットし直す部門が重なる事があり、この時の処理は非常に複雑となる。これは多分最も複雑で長いソートアルゴリズムであろうと筆者は思う。
変更内容、リストは筆者も説明できるほどの理解がなく省略する。また、現状テスト用データは全て処理できたが、本当に完成したアルゴリズムになっているか筆者にはわからない。
後の実測データで示すように比較回数、交換回数は共に 3-1. より増える。速度的には改善されないのではと考えられる。

また8個のデータで最大回数となるのは "7 5 6 2 4 3 1 0", "7 5 6 3 4 1 2 0", "7 6 4 2 5 3 1 0", "7 6 4 3 5 1 2 0" であり、最大時間・平均時間の定式化困難なのも同様である。

ただしデータ4個の時に限ると、固定回数のソートでは比較・交換5回で終わる最も早いソートとなる。4個の部門ではトップとボトムの何れかが入れ替わっても、含む2個の各部門のトップとボトムは変わらず、下層の比較は不要になるからである。
データ8個でも比較・交換が 3-1. より少なく、まだ検討の余地ありかもしれない。

4-6. その他

理論的性能改善はない事は明白である為プログラムを作って確認していないが、場合によっては改善される事もあるアイデアを下記に示す。

部門の割り方として、ここでは連続するデータを同じ部門に割っているが、離れたデータが同じ部門になるように割る事もできる。部門の区切りを添字2進数の下の方から見るのをやめて、例えば上の方から見るのである。
例えば全8個のデータで2個づつの部門に割った時 0b000 と 0b100, 0b0xx と 0b1xx が同じ部門になる。
理論時間自体は変わらないが時間のかかる並び方の分布を変える事ができる。
現実のデータは完全に一様に乱雑ではなく、例えば強いデータ、弱いデータはどこかに固まっている等の偏りはあるものであるが、部門の割り方を変えて調整する事で現実のソート時間を短くできる場合はあるであろう。

トーナメントでデータ末尾側のテイルエンダーから選んでいく方法も考えられる。これは昇順か降順と同じ違いで、方向性を変えるだけでソートの性質は全く変わらない。ただしチャンピオンを選ぶ方ならトーナメント木を決勝までいった時点で、必ずデータの先頭にチャンピオンが入っているが、テイルエンダーを選ぶ方ならデータ個数が 2^N 個でなければテイルエンダーが末尾に入っていない。(範囲外としてスキップされている。)
ソート中にすでにソートされたデータを見たい場合で、最後の方からソートが終わった方が都合が良い場合に対応できる。

5. 速度計測

動作検証、速度計測に使用したプログラムリストは下記の場所に置いてある。

http://boogieshake.web.fc2.com/sort/section_top_sort.c

測定結果は次表に示す。評価はこれまでに記載してあるので省略する。
1 - 12 個までは、全組み合わせを作成してソートしているので、最小/最大/平均は、理論的値と一致する。
100/512/1000 における最小/最大/平均は、いくつか作成したサンプルデータに対する測定値であり、理論的値とは異なる。
gcc のクイックソートの値を一番下の表に示す。ライブラリとしての利用では比較回数しか測定できないが、この時点で負けている事は明らかなので、これ以上の突っ込んだ測定は行わない。

Compare and exchange count measurement in Sorting

Kind	Data amount	Total line	Compare			Exchange		
			minimum	maximum	average	minimum	maximum	average
Section Top Sort [Basic]	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	3	3	3.000000	0	3	1.500000
	4	24	4	6	5.333333	0	6	3.000000
	5	120	8	10	9.333333	0	10	5.000000
	6	720	9	15	13.000000	0	15	7.500000
	7	5040	11	21	18.333334	0	21	10.500000
	8	40320	12	28	23.333334	0	28	14.000000
	9	362880	20	36	31.333334	0	36	18.000000
	10	3628800	21	45	37.666668	0	45	22.500000
	11	39916800	23	55	46.333333	0	55	27.500000
	12	479001600	24	66	54.000000	0	66	33.000000
	100	100	3451	4263	3901.920000	2040	2893	2482.380000
	512	512	65535	107675	102484.134766	60513	70675	65489.023438
	1000	1000	65535	405052	391165.900000	65535	267105	250076.896000
Section Top Sort [Less conflict]	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	3	3	3.000000	0	3	1.500000
	4	24	4	6	5.333333	0	6	3.000000
	5	120	8	10	9.000000	0	8	3.933333
	6	720	9	14	12.033334	0	11	5.900000
	7	5040	11	20	16.995237	0	15	7.677778
	8	40320	12	27	21.298412	0	20	10.465873
	9	362880	20	35	28.061905	0	22	11.408995
	10	3628800	21	39	31.463148	0	27	13.755274
	11	39916800	23	46	37.599262	0	30	15.753829
	12	479001600	24	54	42.750179	0	36	19.101740
	100	100	1074	1249	1161.850000	302	377	336.980000
	512	512	10430	11294	10900.849609	2364	2584	2477.164062
	1000	1000	24502	26346	25445.341000	5061	5421	5245.574000
Section Top Sort [Full exchange]	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	4	5	4.666667	0	6	3.000000
	4	24	4	5	4.666667	0	6	3.000000
	5	120	12	22	18.380952	0	23	11.771429
	6	720	12	22	18.380952	0	23	11.771429
	7	5040	12	22	18.380952	0	23	11.771429
	8	40320	12	22	18.380952	0	23	11.771429
	9	362880		Out of time			Out of time	
	10	3628800		Out of time			Out of time	
	11	39916800		Out of time			Out of time	
	12	479001600		Out of time			Out of time	
	100	100	1112	1618	1358.940000	338	605	469.910000
	512	512	9899	10830	10446.103516	2845	3737	3253.757812
	1000	1000	24211	26457	25560.028000	6060	8135	7197.247000

Kind	Data amount	Total line	Compare			Exchange		
			minimum	maximum	average	minimum	maximum	average
Two Top Sort	1	1	0	0	0.000000	1	1	1.000000
	2	2	2	2	2.000000	1	2	1.500000
	3	6	4	5	4.666667	1	3	2.166667
	4	24	7	7	7.000000	2	7	4.333333
	5	120	10	13	12.000000	1	8	4.733333
	6	720		NG			NG	
	7	5040	14	21	18.009524	2	17	9.357143
	8	40320	17	27	22.842857	3	23	12.766667
	9	362880	26	38	34.001587	4	25	14.512698
	10	3628800		NG			NG	
	11	39916800		NG			NG	
	12	479001600		NG			NG	
	100	100		NG			NG	
	512	512	29664	36859	33064.343750	17361	22830	19948.535156
	1000	1000		NG			NG	
Section Top Sort	1	1	0	0	0.000000	0	0	0.000000
[Set Next]	2	2	1	1	1.000000	0	1	0.500000
	3	6	2	3	2.666667	0	3	1.500000
	4	24	4	6	4.833333	0	6	3.000000
	5	120	5	10	8.000000	0	9	4.733333
	6	720	8	13	10.633333	0	14	7.055556
	7	5040	11	19	14.904762	0	20	9.830159
	8	40320	12	24	18.524603	0	26	13.112698
	9	362880	12	34	25.753483	0	27	13.474956
	10	3628800	17	37	28.135573	0	33	16.468307
	11	39916800	22	43	32.998617	0	41	20.324998
	12	479001600	23	50	37.265566	0	49	24.923850
	100	100	970	1134	1049.19000	402	522	464.920000
	512	512	9772	10588	10194.183594	3241	3681	3429.621094
	1000	1000	22781	24627	23875.151000	6881	7576	7248.754000
Both sides now sort	1	1	0	0	0.000000	0	0	0.000000
	2	2	1	1	1.000000	0	1	0.500000
	3	6	3	3	3.000000	0	3	1.500000
	4	24	5	5	5.000000	0	5	2.666667
	5	120	9	9	9.000000	0	8	4.133333
	6	720	11	13	11.933333	0	12	6.011111
	7	5040	14	18	16.142857	0	15	7.557143
	8	40320	17	23	20.571429	0	19	9.917460
	9	362880	23	33	29.185185	0	22	12.194709
	10	3628800	25	37	32.605926	0	27	14.482081
	11	39916800	28	44	38.304185	0	30	16.427016
	12	479001600	31	53	44.475075	0	37	19.599656
	100	100	1156	1344	1257.900000	320	417	373.320000
	512	512	11786	13679	12766.183594	2627	3207	2920.757812
	1000	1000	28525	32808	30573.943000	5800	7148	6328.246000

Kind	Data amount	Total line	Compare			Exchange		
			minimum	maximum	average	minimum	maximum	average
***gcc q-sort**	1	1	0	0	0.000000			
(For comparison)	2	2	1	1	1.000000			
	3	6	2	3	2.666667			
	4	24	3	6	4.916667			
	5	120	4	10	7.716667			
	6	720	5	15	11.050000			
	7	5040	6	21	14.907143		Can’t count	
	8	40320	14	29	19.171354			
	9	362880	16	36	22.888112			
	10	3628800	18	47	27.061262			
	11	39916800	20	55	31.299797			
	12	479001600	22	70	35.746590			
	100	100	603	699	641.710000			
	512	512	4416	5064	4588.541016			
	1000	1000	9661	10708	10003.768000			

6. 結論

本稿では、筆者が考え出したソートアルゴリズムの基本となるものと、様々な改造による速度改善の試みを述べた。
従来発表された物でない可能性があり、部分ソート、並列処理ソート等、有用である可能性があると筆者は思うので、ここに文書化する次第である。
今後の展開としては、まずこのアルゴリズムの最速版での最大時間、平均時間を推測でなく、確証を得る事が必要であろう。
また、並列処理化との相性が良いと述べたことに対して、実際に並列処理化したアルゴリズムの作成・提示が必要であろう。

7. 参考文献

筆者は研究職でなく論文の閲覧が簡単にできる立場に無いので参考文献はネット上で簡単に検索できる物のみに止まる。
現在知られるソートの種類およびソートアルゴリズムの性質を示す分類の用語は 1) を参照した。
またコーネル大学のアーカイブでソートを表題に含むものを検索し、4) 5) からソートアルゴリズムの最近の利用動向を確認している。

1) Wikipedia "Sorting algorithm" article and all its related article
2) Wikipedia "Tournament sort" article
3) arXiv.org > cs > arXiv:1402.2712 "Dynamic Partial Sorting"
4) arXiv.org > cs > arXiv:1609.04471 "Sort Race"
5) arXiv.org > cs > arXiv:1511.03404 "Comparison of parallel sorting algorithms"