Pbit and other list sorting algorithms

David S. Płaneta*

January 17, 2006

Abstract

Pbit, besides its simplicity, is definitely the fastest list sorting algorithm. It considerably surpasses all already known methods. Among many advantages, it is stable, linear and be made to run in place. I will compare Pbit with algorithm described by Donald E. Knuth [8] in the third volume of "The Art of Computer Programming" and other list sorting algorithms.

1 Introduction to lists

A Lists is a set of independent data formats, very often called nodes. Particular nodes are connected with each other by means of pointers. Nodes are usually created dynamically, which is good for economical use of memory. We will deal with singly-linked lists because Pbit was created for such type of lists (which does not prevent from sorting double-linked lists). In a singly-linked list each node contains pointer only for another node in the list. The last pointer shows NULL. To remember at which point the list starts there is a special pointer called root or a head of the list [1]. The pointer of root may also be the first node of the list. Queue or Stack are examples of a singly-linked list.

2 Structure of data representing a node and description of a simply function inserting into the singly-linked list.

```
struct Node{
Node *next;
TYPE data;
};
```

A node is the structure which contains at last one pointer for 'node' and field of any 'data' type, on the ground of which the nodes will be sorted.

We know what a particular of the list looks like, so let us analyse the function inserting elements into it. Function '`push`' corresponds with the function stacking elements.

*dplaneta@gmail.com

```
1 void push(Node **root, TYPE value){
2 Node *new_Node = new(std::nothrow) Node;
3 if(new_Node == NULL){
4 //Error – memory overflow.
5 return;
  }
6 new_Node->data = value;
7 new_Node->next = *root;
8 *root = new_Node;
  }
```

1. Function '`push`' takes two parameters. One is the pointer of the first element of the list (root) and the other argument is the value added to the list.
2. We declare pointer of '`node`' type, next we attribute to it memory declared by the '**new**' operator.
3. If the pointer shows NULL it means there is no space in memory (memory overflow).
4. Function should send the message about an error and ...
5. Finish working.
6. Haring dealt with storage allocation, we write in the new node the value which was passed by function argument.
7. Node '`next`' is ascribed address of the root (head) that is attributed to first element of the list.
8. The root indicate at the new node.

3 The idea of sorting using Pbit

The whole idea of Pbit is very interesting. Besides its simplicity it is indisputable the fastest algorithm which sorts lists. Among many advantages, it is stable, linear and does not require memory ("in situ" sorting algorithm).

Split the list in relation to the bit pattern. Each pattern variation is represented by separate list. Apply recurrently the same sorting function to all the lists but remember to project the pattern onto the succeeding bits. When the last key is looked through, merge *sorted fragments* of the list.

Pbit is combination of BucketSort [2], RadixSort [3] and MergeSort [4] ideas.

Bit pattern is a set of $\boldsymbol{K}$ bits. $\boldsymbol{K}$ defines the number of bits which are cut off the

binary word. $\boldsymbol{M}$ defines number of bits in a binary word. For example, one-byte sign type 'char' contains 8 bits. Four-byte number is 32 bits long. Ω is variation $\boldsymbol{K}$ of element binary set $\{0,1\}$, $\Omega = 2^K$. It determines the number of groups (number of roots), which the list may be divided into during one step. $\boldsymbol{n}$ is the number of sorted elements. Meaningless for linear Pbit sorting. Coefficient "by $\boldsymbol{n}$" depends only on $\boldsymbol{K}$ (length of bit pattern) and $\boldsymbol{M}$ (number of bits in binary word). Coefficient "by $\boldsymbol{n}$" can be determined by dividing $\boldsymbol{M}$ by $\boldsymbol{K}$. If using my merging method, add $\boldsymbol{n}$ to the coefficient.
$f(n) = \frac{M}{K} * n + n$
In Pbit I use four-bit pattern ($K = 4$) for short lists, which means that I split the list into sixteen groups in maximum($\Omega = 2^K = 2^4 = 16$).
Constants in (source code) Pbit sorting algorithm:
$\boldsymbol{K} = 4$ (four-bit pattern is excellent for short lists sorting)
Ω - dependent on $\boldsymbol{K}$; equals ($2^K = 16$)
Variables (depending on type of data):
$\boldsymbol{M}$ - depends on the size of sorted element.
$\boldsymbol{n}$ - depends on the number of elements.
Example: We have a list consisting of eight one-byte elements:
$21 \longrightarrow 3 \longrightarrow 209 \longrightarrow 14 \longrightarrow 156 \longrightarrow 47 \longrightarrow 3 \longrightarrow 214 \longrightarrow$ NULL
We put element of the list in relation to the first four bits ($K = 4$) from the right (the most important – little-endian order of byte):
{3,14,3} — {21} — {47} — {156} — {209,214}
Next, we put elements in each groups separately looking at the following four bits (in case of one-byte type - the last ones):
{ {3} — {3} — {14} } — {21} — {47} — {156} — { {209} — {214} }
In "*two steps*" $\boldsymbol{n}$ one-byte elements are sorted.

In how many steps four-byte numbers will be sorted?
In $\frac{M}{K}$. Number of bits of sorted element divided by the length of bit pattern. The length K of the pattern is constant which equals 4. In case of four-byte elements variable $\boldsymbol{K}$ equals 32 (because 4 bytes = $4 * 8$ bits = 32) bits. So coefficient "by $\boldsymbol{n}$" equals $\frac{32}{4} + 1 = 8 + 1 = 9$. In eight steps four-byte numbers are sorted. In one step the lists are merged.

O(n) algorithms such as countsort or radixsort required memory proportionally to **n**. *How is it possible that* Pbit does not require memory proportionally to **n***?*
Pbit does not count how many times number 'x' occurred in sequence 'L'. It does not compare numbers, (with one another) either. This algorithm is based on **a *specific grouping* of elements** and needs only a few bytes parameters or roots. It is a very little amount of memory independent of $\boldsymbol{n}$ — number of elements to be sorted. Precisely, it depends on Ω and coefficient.
The maximum amount of occupied storage can be expressed by formula:
$T = (\Omega * 4 + 3 * 4) * \frac{M}{K} + 2 * 4$ bytes, assume size of pointer equal four bytes
For constant $\boldsymbol{K}$ which equals four, Ω equals sixteen.
$T = (64 + 12) * \frac{M}{4} + 8$ bytes
$T = 19 * M + 8$ bytes

The amount of occupied storage ***T(M)*** will depend only on the size of sorted element.
For sign types (1 bytes = 8 bits) Pbit will require maximum $T(8) = 160$ bytes.
For "integers" (4 bytes = 32 bits) Pbit will require maximum $T(32) = 616$ bytes.
It means that it needs constant, modest space usage.
Space usage $= \Theta(\frac{M}{K} * \Omega)$
$\Omega = 2^K$, K - constant
M defines number of bits in a binary word (size of TYPE). There is M constant.
That's why memory complexity equal O(1).
The greatest advantages of Pbit is it be made to run in place.
It requires merely O(1) memory!

Why maximum instead of constant amount of memory?
Because "empty" lists are omitted and are not taken into account during successive recurrent calls.

What order algorithm is responsible for merging list?
It's O(n), because Pbit merges only those fragments of the list which were sorted. We group list elements till the last key. When the last key is looked through, all we have to do is to merge sorted fragments of the list. Every few, maximum $\frac{M}{K}$, recurrent calls sorted fragment of the list is (recursion) merged.

Is linearity the best advantage of Pbit?
I don't think so. O(n) time algorithms rather will be much faster than O(nlgn) only while sorting very long lists.
The greatest advantage of Pbit does not require memory proportionally to ***n*** (it's not extensive). Pbit is stability and the fact that it can be adapted for sorting "special data formats". In case of Pbit you can't talk about expected or pessimistic time complexity. Time complexity is always the some, no matter how many elements there are and what arrangement of sequence is (sorted, all elements are the some). In the four position I would put the statement about linear time complexity, small coefficient. Last, but not last advantage of Pbit is simplicity in implementation and clear idea.

4 Cutting bits off in binary words

All quantities in computer are represented by binary words.

Pbit is based on cutting bits off and sorting elements in relation to those bits. Unfortunately, not every high-standard language has built-in operators which carry out operations on bits. My means of arithmetic operators, however, it's possible to build a function which can cut bits off in a binary word. Pbit will use function of cutting bits off in a binary word, which in Pascal language can be defined as follows:

```
function bits(number, M: integer) : integer;
begin
        bits := (number div ( 2^M ) ) mod Ω
end
```

M - bit number, from which the pattern in binary word is determined
$\Omega = 16$

Dividing by power of two is like shift bits to the right with index exponent.
For example, we divide $123/8 = 15,375$
123 decimally is 01111011 binary.
We move bits three positions to the right (three because the third power of two equals eight): 00001111 binary is 15 decimally. In Turbo Pascal there is a special function for this purpose.
Language C has built-in operator "`>>`".

`number div` 2^M
in C can be written:
`number >> M`

The result is divided by modulo Ω. The operation of modulo division is carry in order to read only ***K*** last bits ($\Omega = 2^K = 16$). The first versions of algorithm for cutting off bits in sorted elements used projection with bit field structure. To implement algorithm in other program languages, not only C/C++, I have chosen arithmetic/logic method.

`bits := (number div (`2^M`)) mod` Ω
In C operations div and modulo can be changed into faster logical operations:
`bits = (number >> M ) & ` $(\Omega - 1)$, assume $\Omega = 2^K$ $(\Omega - 1)$ - received number should be written hexadecimals.

I use bit shift, instead of division by power of two and logical operator '`AND`', instead of modulo division because the statement which counts index of root table should be optimized to a maximum. It is this statement that the speed of function depends on.

5 Implementation in C++

```
1  Node *Pbit(Node *L, unsigned M, Node *P=NULL){
2  if(L){
3        M=M-K;
4        Node *tab[Ω]={NULL};

5        for(Node *i,**in; i=L; i->next=*in,*in=i){
6        in=&tab[(L->data>>M) % Ω];
7        L=L->next;
         }

8        if(M) for(int i=0; i<Ω; i++) P=Pbit(tab[i], M, P);
9        else for(int i=0; i<Ω; i++) P=merge(tab[i], P);
         }

10 return P;
   }
```

Function takes parameters:

L – the first node of the list to be sorted.
P – "end maker", assumptive equals NULL.
M – size of sorted element expressed in bits.

Function returns pointer to the first element of the sorted in decreasing order.

1. `Node *Pbit(Node *L, unsigned M, Node *P=NULL);`
   Function has three parameters, one of which assumptive equals NULL. Two parameters and function one of Node. Node is the structure which consists of at least pointer of 'Node' type and variable 'data'. List will be sorted in relation to variable (key) 'data'. There is a broader description of lists and structure in the first and second point

2. After calling the function we check whether the list which Pbit is to sort is not empty. Such check-up is not necessary but it makes the algorithm faster, because empty lists are omitted in the process of grouping. If the list happens to be empty we return pointer P (end marker), thus we return part of a sorted list. Additional, we could check if we do not sort one-element list because in case of one-element list it's not necessary to look through all bits of sorted element. However, it does not bring any significant benefits.

3. Operation of subtraction of pattern length from length of sorted element is connected with subsequent grouping of nodes. At the first stage of sorting for four-byte numbers $M = 32$ and constant $K = 4$. $M = M - K$, $M = 28$ – which means that we will read determine list number into with 28 (to 32) bit. Four bits which were read determine list number into which we insert sorted element.

4. We create root table of new lists. Sorted nodes will be placed in those lists. Each elements of table is declared with NULL. `table[size]={0,...}` - such initiation of table elements is correct. Null value will be attributed to each element. Compiler may signal lack of initiation values.

5. Pointers declared in loop 'for' work similarly to function inserting new elements in the list. The description of algorithm responsible for this process is in the second point.

6. Table index is four-bit pattern received after perusal of $K = 4$ bits starting with M bit of binary word 'data'. Index value is `(L->data>>M) % Ω`, it's better to replace modulo operation with logical 'AND' operation `(L->data>>M) & (Ω-1)`, because probably it's five times as fast as remainder of division operator. Such efficiency jump is very important in a loop which decides about the speed of algorithm. If we leave modulo operator, it's possible that compiler will change modulo operator into logical 'AND' during the process of optimization . Additionally, it's possible to make indexing more optimum:
   Instead: `in=&tab[ (L->data>>M)&0xf ]`, we can write
   `in=tab+((L->data>>M)&0xf)`
   Because 'in' is of "pointer to pointer" type.
   `&tab[index] = &(*(tab+index)) = tab + index`

7. After inserting node into a new list, pointer 'L' will show another element in the list.

8. In these two lines we decide if we will carry on sorting or if it's high time to merge (sorted) lists. If 'M' is greater than zero, it means that we haven't looked through all bits yet. This is why we sort all newly created lists in relation to those non-locked through bits – each separately.

9. If 'M' equals zero it means that we have looked through all bits of sorted numbers. Then, we start the process of list merging (function merge and how it works will be described on the next section).

10. If the sorted list was empty we return end-maker 'P' (NULL) stated in parameter. If not we return "pointer P" – indicating the sorted list.

5.1 Function merge and problem of list merging.

Function merge was adjusted to the root type. Roots in singly-linked lists can be divided into two groups. Roots of the first group consist of two pointers. One pointer indicates the first element of the list while the other one indicates the last element. The second list contains roots which are the first elements of lists. They are nodes. I have chosen the root which represents the second group approach. Function merge(A,B) merges list 'B' with the end of list 'A'. Returns address as the first node of a new list. To do that it goes through the whole list 'A' and changes bridging of the last link 'NULL' into 'B'. Anyone who knows data structures very well (which singly-linked list are) may claim that it would be better to use lists of the first root type. Always, independently of $\boldsymbol{n}$ We would make one operations. Using the second type node, however, enables us to make $\boldsymbol{n}$ operations. Looking at sorting as a whole the second way will be faster. *Why?* In Pbit we split the list more often that we merge it. By using the second type roots we put successive elements to the lists faster.

Definition of function responsible for list merging:

```
1 Node *merge(Node *A, Node *B){
2 if(A==0) return B;

3 Node *temp=A;
4 while(temp->next) temp=temp->next;
5 temp->next=B;
6 return A;
  }
```

1. Assumption: We insert node ‘A’ at the end list ‘B’.
2. If list ‘A’ is empty, we transmit list ‘B’.
3. Temporary variable ‘temp’ remembers address of the first node of list ‘A’.
4. We move to the last element of list ‘A’.
5. We change bridging of the last element of the list ‘A’ : NULL into ‘B’.
6. We return the first element of list ‘A’.

Wouldn’t it be better to check first it list ‘B’ is empty?
Why making loops and going through to the last element of list ‘A’ since in case ‘B’ = NULL we change nothing? No, it wouldn’t, because in most cases list ‘B’ will not be empty. Additional statement “if” would make time of list merging worse.

You might point out that execution time is $\Theta(n * \frac{M}{K})$, since there are $\frac{M}{K}$ steps and each of $\boldsymbol{n}$ items is examined in each step. At worst, the merging takes another $\boldsymbol{n}$ steps to get to the end of each list.
Of course no! For example include counter on merge function:

```
global unsigned count;

Node *merge(Node *A, Node *B){
if(A==0) return B; //we do nothing
count++; //first node (maybe only)
Node *temp=A;
while(temp->next){
temp=temp->next;
count++; //n-node of list
}
temp->next=B;
return A;
}
```

We group list elements till the last key. When the last key is looked through, all we have to do is to merge sorted fragments of the list. Every few, maximum $\frac{M}{K}$, recurrent calls sorted fragment of the list is (recursion) merged. Pbit by turns merges and splits the lists. *It's* $(n * \frac{M}{K}[split]) + n[merge]$ *not* $(n * \frac{M}{K}[split]) * n[merge]$ **Always *count* equal *n*** Only that it is linear and non-extensive.

6 Differences between sorting in descending and ascending order

In case of sorting in decreasing order ($10 \to 9 \to 8 \to \ldots \to$ NULL) lists are merged from the root which represents the smallest elements till the root which represents the greatest ones and the greatest element (node address) is returned.
In case of sorting in ascending order ($8 \to 9 \to 10 \to \ldots \to$ NULL) lists are merged from the greatest to the smallest one. The smallest element is returned.

```
1   Node *Pbit(Node *L, unsigned M, Node *P=NULL){
2   if(L){
3          M-=K;
4          Node *tab[Ω]={NULL};

5          for(Node *i,**in; i=L; i->next=*in,*in=i){
6          in=tab + ((L->data>>M) & (Ω-1));
7          L=L->next;
           }

8          if(M) for(int i=0; i<Ω; i++) P=Pbit(tab[i], M, P);
9          else for(int i=0; i<Ω; i++) P=merge(tab[i], P);
           }

10  return P;
    }
```

(8) and (9) line in sorting code undergo change.

In case of sorting in decreasing order:

8. **if**(M) **for**(**int** i=0; i<Ω; i++) P=Pbit(tab[i], M, P);

9. **else for**(**int** i=0; i<Ω; i++) P=merge(tab[i], P);

We begin to split and merge list from tab[0] (list containing the smallest elements) and end with the last list (containing the greatest elements). Pointer ‘P’ indicates the node with the highest value.

In case of sorting in ascending order:

8. **if**(M) **for**(**int** i=($\Omega - 1$); i>=0; i--) P=Pbit(tab[i], M, P);

9. **else for**(**int** i=($\Omega - 1$); i>=0; i--) P=merge(tab[i], P);

We begin to split and merge lists from tab[$\Omega - 1$] (list containing the greatest elements), and end with the last list (containing the smallest elements). Pointer ‘P’ indicates the node with the lowest value.

7 Negative numbers sorting

So far we here discussed integer non-negative numbers sorting. To sort the list containing numbers “with sign” we use not one but two call a function Pbit. This is why sorting function should require information if the sorted numbers are “with sign” or unsigned. It should also take information how to sort - in ascending or decreasing order.

```
Node *Pbit(Node *L, int sign_numbers=1, int sort_decreasing=0);
```

We assume that marker `sign_numbers = 1` and `sort_decreasing = 0`.

`sign_numbers = 1`
It means that we sort numbers “with sign”.

The problem with ‘`sign_numbers`’ definition we may leave Pbit function or use “Run-Time Type Identification” from `typeinfo.h` library.

```
template<class type> inline int sign_numbers(type){
return (type)((type)1<<((sizeof(type)<<3)-1))<0;
}
```

Only for integer the function is able to do!

`sort_decreasing = 0`
It means that we sort in ascending order.
Each value than ‘0’ indicates sorting in ascending order.

Pbit written in pseudo-code

```
if(sign_numbers){
Sorts numbers 'with sign' (positive and negative)
if(sort_decreasing) Sorts in decreasing order
else Sorts in ascending order
}
else{
Sorts numbers unsigned (non-negative)
if(sort_decreasing) Sorts in decreasing order
else Sorts in ascending order
}
```

Numbers “with sign” and non-negative (unsigned) are recorded in a different way. For representing numbers “with sign” so called “binary complement” is use. To understand fragment of the code responsible for sorting numbers “with sign” we must only know that the most important bits of negative numbers are always greater than the most important bits of positive numbers. This is why while sorting negative numbers using Pbit, which sorts in decreasing order, in fact receive ascending list. There is a complete Pbit code written in C++ on another section.

8 Complete Pbit implementation in C++

```
Node *Pbit(Node *L,int sign_numbers,int sort_decreasing);
if(L==0) return 0;
unsigned size=sizeof(L->data)<<3;
//sign_numbers = (1)-sort numbers 'with sign' (negative)
//                (0)-unsigned (non-negative)
if(sign_numbers){
Node *L_unsigned = 0, *L_negative = 0;
//split list into negative and
//non-negative (unsigned) numbers
for(register Node *temp; temp = L; ){
L = L->next;
if(temp->data < 0){
temp->next = L_negative; L_negative = temp;
}else{ temp->next = L_unsigned; L_unsigned = temp; }
}
return (sort_decreasing)?
Pbit_decreasing(L_unsigned, size,
                Pbit_decreasing(L_negative, size) ):
Pbit_ascending(L_negative, size,
              Pbit_ascending(L_unsigned, size) );
}
else return (sort_decreasing)? Pbit_decreasing(L, size):
                               Pbit_ascending(L, size);
}
```

`Pbit_decreasing` – we sort list in decreasing order (returns pointer to the greatest element).
`Pbit_ascending` – we sort list in ascending order

(returns pointer to the smallest element).

2. We check if sorted list ‘L’ is not empty.
3. We determine the size of sorted element in bits.
4. If (logical) variable `sign_numbers` if different than zero, it means that sorted elements are signed.
5. We create two pointers which will become the roots of new lists; list representing negative numbers and list representing non-negative (unsigned) numbers.

6-10. We split lists, list roots represent nodes (5).

11. If variable `sort_decreasing` is…
12. different than zero we sort in decreasing order ($10 \rightarrow 9 \rightarrow 8 \rightarrow \ldots$),
13. if not we sort in ascending order ($8 \rightarrow 9 \rightarrow 10 \rightarrow \ldots$).
14. Variable `sign_numbers` is equal to zero. It means that we sort numbers “unsigned” (non-negative).

14-15. If variable `sort_decreasing` in different than zero we sort in decreasing order, if not we sort in ascending order.

9 Comparison of list sorting algorithms

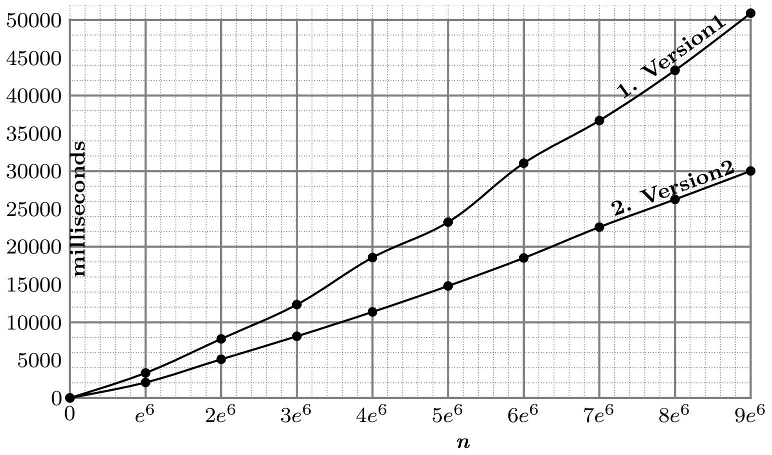
QuickerSort [5] is characterised by very small O(nlgn) coefficient. The number of other algorithm operations is not significantly greater than number of comparison. QuickerSort is based on list division in relation to the first node (root). We divide the list into elements which are equal, greater or less than the chosen node. Next, the “biggest” (by mean of value) and the “smallest” lists are recurrently sorted. Its most serious disadvantages are instability and cost $\mathrm{O}(n^2)$ in worst case behavior. In storage complexity we must take into account memory for stack and heap implementation. As far as worst case behaviour is concerned, the depth recursion is $n-1$.

Version 1

```
Node *merge(Node *t, Node *P){
if(t==0) return P;

Node *tmp = t;
while(tmp->next) tmp=tmp->next;
tmp->next=P;

return t;
}


Node *qs(Node *L, Node *P=0){
if(L==0) return P;

Node *L1=0, *L2=0, *L3=0;
L2=L; L=L->n; L2->next=0;

for(Node *i;i=L;){
L=L->next;
if(L2->data > i->data)
{ i->next=L1;  L1=i; }
else if(L2->data == i->data)
{ i->next=L2;  L2=i; }
else
{ i->next=L3;  L3=i; }
}

return
merge(qs(L3),merge(L2,qs(L1)));
}
```

Version 2

```
Node *qs(Node *L, Node **P=0){
if(L==0) return 0;
if(L->next == 0) return (*P = L);

Node *L1=0, *L2=L, *L3=0;
Node *P1=0, *P2=L, *P3=0;

L=L->next;
L2->next=0;
for(Node *i;i=L;){
L=L->next;
if(L2->data < i->data)
{ i->next=L1; L1=i; }
else if(L2->data == i->data)
{ i->next=L2; L2=i; }
else
{ i->next=L3; L3=i; }
}

L3=qs(L3, &P3);
if(L3) P3->next=L2;
else L3=L2;
if(L1){
P2->next=qs(L1, &P1);
if(P) *P=P1;
}
else if(P) *P=P2;
return L3;
}
```



n	e^6	$2e^6$	$3e^6$	$4e^6$	$5e^6$	$6e^6$	$7e^6$	$8e^6$	$9e^6$
1.	3305	7811	12348	18567	23253	31045	36693	43333	50893
2.	2043	5107	8152	11376	14811	18516	22593	26257	30025

1. QuickerSort (with outside the merge function) Version1.

2. QuickerSort (without the merge function) Version2[1].

MergeSort [4] like QuickerSort is based on element comparison. The best description of MergeSort is based on recursion. Algorithm divides list into two lists of the some size. Next, both lists are sorted (MergeSort) separately. To finish the process of sorting original n element list, two sorted halves are merged. Unfortunately, recurrent implementation of algorithm does not sort "in situ", so we need twice as big memory as occupied by unsorted data. On my hardware version1 of algorithm [6] stopped list sorting which consisted of over million elements – it overflow the stack. It considerably restricts usefulness of "advanced recurrent" implementation of algorithm. This is why I used to comparision ***partly iterative***(version2), more efficient MergeSort implementation.

[1] To make a comparison I used more efficient version2.

Version 1	Version 2

Version 1

```
Node *MergeSort(Node *);
Node *merge(Node *, Node *);
Node *split(Node *);


Node *MergeSort(Node *list){
if(list==0) return 0;
if(list->next==0) return list;
Node *SecondList=split(list);
return merge(MergeSort(list),
MergeSort(SecondList));
}


Node *merge(Node *list1,
Node *list2){
if(list1==0) return list2;
if(list2==0) return list1;
 if(list1->data <= list2->data){
 list1->next=
  merge(list1->next, list2);
 return list1;
 }
 else{
 list2->next=
  merge(list1,list2->next);
 return list2;
 }

}


Node *split(Node *list){
if(list==0) return 0;
if(list->next==0) return 0;

Node *pSecondCell=list->next;
list->next=pSecondCell->next;
pSecondCell->next=
 split(pSecondCell->next);
return pSecondCell;
}
```

Version 2

```
Node *MergeSort(Node *);
Node *merge(Node *, Node *);
Node *split(Node *);


Node *MergeSort(Node *list){
if(list==0) return 0;
if(list->next==0) return list;
Node *SecondList=split(list);
return merge(MergeSort(list),
MergeSort(SecondList));
}


Node *merge(Node *list1,
Node *list2){
 if(list1 == 0) return list2;
 if(list2 == 0) return list1;

 Node *list = 0, *last = 0;
 while(list1 && list2){
 Node* p;
 if(list1->data <= list2->data){
 p = list1; list1 = list1->next;
 }else{
 p = list2; list2 = list2->next;
 }
 if(list) last = last->next = p;
 else last = list = p;
 }
 if(list1) last->next = list1;
 else last->next = list2;
 return list;
}


Node *split(Node *list)
{
 Node* list2 = 0;
 while(list && list->next){
 Node* w = list->next->next;
 list->next->next = list2;
 list2 = list->next;
 list->next = w;
 list = list->next;
 }
 return list2;
}
```

Unfortunately, recursion MergeSort implementation overflows the stack by sorting over-million list. For such list it obtained time 6569 (version2: 4016) milliseconds.

I proposed **Psort** instead of QuickerSort to lists sorting. Algorithm divides the list into three sub-lists in relation to two nodes, into elements less than the first node, greater than the first node but less than the second node and greater than the second node. It's excellent for sorting short lists.

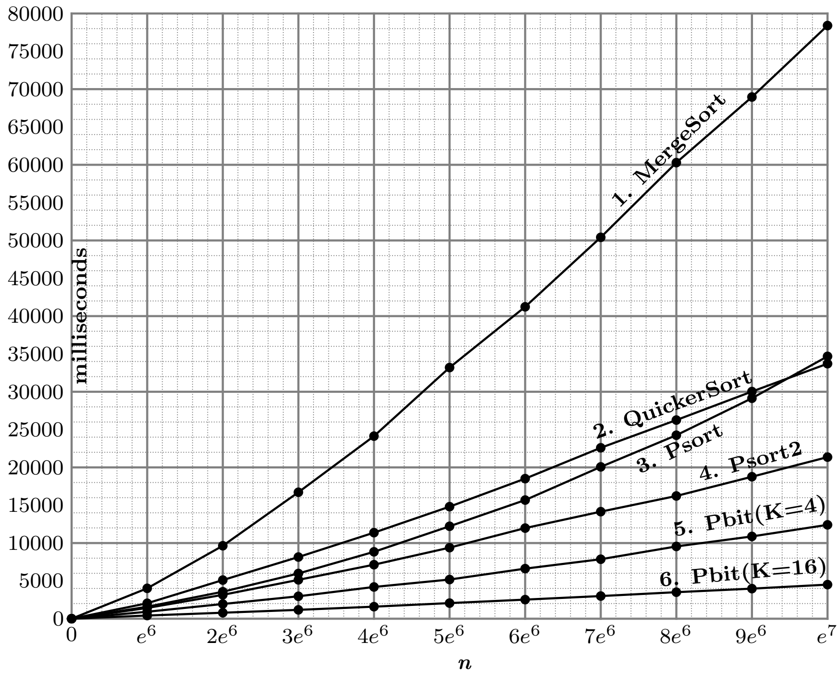
```
Node *Psort(register Node *L, Node *P=0){
if(L==0) return P;
if(L->next==0) return (L->next=P,L);

Node *P2, *L2=0;
{
Node *i,
*tmp=(L->data > L->next->data? P2=L,L=L->next : P2=L->next)->next;

for(P2->next=L->next=0; i=tmp; ){
tmp=tmp->next;
if(i->data < L->data){ i->next=L->next; L->next=i; }
else if(i->data > P2->data){ i->next=L2; L2=i; }
else{ i->next=P2->next; P2->next=i; }
}

}

L->next=Psort(L->next,P);
P2->next=Psort(P2->next,L);
return Psort(L2,P2);
}
```

Psort2 is modification of **Psort**. I implemented it for sorting long lists with often repeated value. Wegner [7] describes similar ternary partitioning algorithm more efficient than QuickSort.

```
Node *Psort2(Node *L, Node *P=0){
if(L==0) return P;
if(L->next==0) return (L->next=P,L);

Node *P2, *L2=0;
{//---2
Node *PP;

{//---1
Node *LL;

{//---0
 Node *i,
 *tmp=(L->data > L->next->data? P2=L,L=L->next : P2=L->next)->next;

 for(PP=P2, LL=L, P2->next=L->next=0; i=tmp; ){//_for
  tmp=tmp->next;
  if(i->data < L->data){ i->next=L->next; L->next=i; }
  else if(i->data > P2->data){ i->next=L2; L2=i; }
  else{//_else1
   if(i->data == L->data){i->next=LL; LL=i;}
   else if(i->data == P2->data){i->next=PP; PP=i;}
   else{i->next=P2->next; P2->next=i;}
  }//else1_
 }//for_
}//---0

L->next=Psort2(L->next,P);
L=LL;
}//---1

P2->next=Psort2(P2->next,L);
P2=PP;
}//---2

return Psort2(L2,P2);
}
```

Pbit sorts by grouping nodes. It belongs to the group of algorithms which I worked out and which uses "end marker" 'P'. It is linear sorting algorithm with coefficient dependent on the length of pattern '$\boldsymbol{K}$'. Conventionally pattern was set up for four bits. It makes sorting both short and long lists effective. It also has influence on very little need for memory, depending on size of sorted elements and not their number (it's non-extensive). Each programmer may change the length of the pattern and adept Pbit for his use. Such problems as "worst case behaviour" do not occur in case of Pbit. Data base programmers don't have to create a few lists, attribute them to particular data in structure and sort every list separately owing to stability of the algorithm. Above fact show why Pbit is considered not only as the fastest but also the most suitable for lists sorting.

Tests were carried out on computer equipped with processor AMD Athlon XP 1800+ (1533 Mhz). For needs of tests I filled the list with pseudo-random numbers obtained from function rand(), `RAND_MAX=0x7FFF`, from standard library C. Each time I initiated current time into pseudo-random numbers generator. Every test was carried out ten times, arithmetic mean of obtained results was written in the table.



n	e^6	$2e^6$	$3e^6$	$4e^6$	$5e^6$	$6e^6$	$7e^6$	$8e^6$	$9e^6$	e^7
1.	4016	9644	16714	24134	33198	41229	50412	60287	68959	78403
2.	2043	5107	8152	11376	14811	18516	22593	26257	30025	34410
3.	1582	3565	5979	8842	12218	15683	20069	24254	29142	34680
4.	1422	3155	5137	7141	9393	11977	14150	16223	18767	21361
5.	921	1943	2964	4196	5178	6619	7862	9553	10876	12407
6.	421	791	1171	1592	2063	2524	2995	3505	3976	4506

Unfortunately, recursion MergeSort implementation overflows the stack by sorting over-million list. For such list it obtained time 6569 milliseconds. I used iterative MergeSort implementation in the test.

For ten-million (long) list Pbit sorting with sixteen-bit pattern is more than **seventeen times faster** than the most popular MergeSort!

9.1 Point on axis n (number of elements) in relation to linear-logarithmic order

Probably each linear algorithm has some point on axis n (number of elements) in relation to which it is slower than linear-logarithmic algorithm. Let's check from which point on axis O(n) Pbit is slower than QuickerSort O(nlgn).

For bit pattern which equals four we sort n four-byte natural numbers:
For $K = 4$ bits, $M = 32$ bits

$\Theta(8n)$ will be faster than $\Theta(nlog_2 n)$ for $n > 256$ elements.
However we cannot compare only coefficients since the most important operations of algorithms differ in time. In respect of time QuickerSort differs from Pbit on statement responsible list division:

QuickerSort	Pbit

if(data>x)
else if(data==x)
else
	tab[(data>>M) & 0xF]
	0xF = $(\Omega - 1)$ – example constant

At the average n elements will be compared $1.5nlog_2 n$ times	In case of Pbit n elements (relations) will be shifted $(M/K) * n$ times
Now we will analyse time consumption of operations. Because all ticks of the clock are the same, we omit them and give only mean time in nanosecond.	
Operation "if" in C requires 20 nanosecond on the average	Operations of bit shift, logical 'AND' and reference to table element take in total 45 nanoseconds on the average in C
n elements sorted in $20 * 1.5 * nlog_2 n = 30nlog_2 n$ nanoseconds	n elements sorted in $45 * 8n = 360n$ nanoseconds

$$30nlog_2 n > 360n, n > 0$$
$$nlog_2 n > 12n$$
$$log_2 n > 12$$
$$n > 4096$$

For $n > 4096$ elements Pbit which splits thirty two-bit numbers with four-bit pattern will be faster.

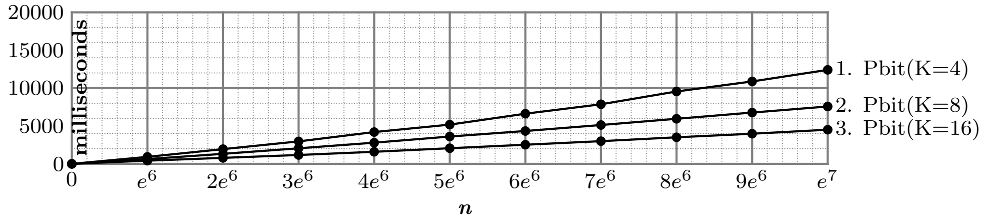
It does not mean, however, that we will see the differences while sorting less than 4096 elements. Modern computers will need only one hundredth of second to sort (linear-logarithmic) even ten thousand elements. ***It means that we should always use Pbit for lists sorting.*** For long lists we can enlarge the length of the pattern. It will make the coefficient smaller and consequently the algorithm will be faster. For example, for $K = 8$, coefficient equals four, which in result will cause efficiency jump and lessen the number of nodes, which linear Pbit will always be faster than linear-logarithmic algorithms to sixty four elements!

10 The length of bit pattern

For longer pattern algorithm will require more memory.
Maximum amount of occupied storage can be expressed by a formula: $T = (\Omega * 4 + 3 * 4) * \frac{M}{K} + 2 * 4$ [byte]
If $K = 4$, algorithm will need 616 bytes to sort four-byte numbers. Whereas for $K = 8$, algorithm will require as much as 4152 bytes.

Constants: $K = 8$, $\Omega = 256$, $\text{hex}(\Omega - 1) = \text{hex}(256 - 1) = \text{hex}(255) = 0\text{xFF}$

```
Node *Pbit(Node *L, unsigned M, Node *P=NULL){
 if(L){
          M-=8;
          Node *tab[256]={NULL};

          for(Node *i,**in; i=L; i->n=*in,*in=i){
          in=tab + ( (L->data>>M) & 0xFF );
          L=L->next;
          }

          if(M) for(int i=0; i<256; i++)  P=Pbit(tab[i], M, P);
          else   for(int i=0; i<256; i++)  P=merge(tab[i], P);
         }

 return P;
}
```



n	e^6	$2e^6$	$3e^6$	$4e^6$	$5e^6$	$6e^6$	$7e^6$	$8e^6$	$9e^6$	e^7
1.	921	1943	2964	4196	5178	6619	7862	9553	10876	12407
2.	631	1322	2053	2794	3605	4336	5127	5948	6770	7571
3.	421	791	1171	1592	2063	2524	2995	3505	3976	4506

11 Differences between Pbit and other algorithms sorting with "*bit key*"

I will compare Pbit with algorithm described by Donald E. Knuth [8].

Address counting sort
In case of Pbit there is no worst case. If all elements are identical, it takes far less time to sort the list than for non-uniform distribution of the keys.
Position interchange
Position sorting by swamping can be used to sort mainly tables. Pbit does not look through the both sides of the list.
Scatter sort
Recurrent scatter sort, unlike Pbit, has very high proportionality factor.
List position sorting
Pbit scatters in relation to the most significant bits, not to the least significant ones. We scatter list elements till the last bit and lists scattered that way are merged. Only sorted of the list are merged.
Position Sort in relation to the most significant number
Algorithm spends too much time on small "chunks". Algorithm sorting in relation to the least significant numbers is relatively efficient. Probably the best compromise was suggested by M.D. MacLaren in 1966 [9]. He recommended using sorting with the least significant number as first but use it only for most significant numbers. After that sequence will not be completely sorted but almost ordered, so not much inversion will be left there. This is why it is possible to finish sorting by using simple insertion. Also W. Dobosiewicz [10] dealt with algorithm sorting with the most significant number as first until he obtained short subsequences. Unfortunately, in case of non-uniform key arrangement, algorithm has worst case O(nlgn). The work of Dobosiewicz inspired some other scholars to invent new algorithms based on address counting the best known of which is perhaps the scheme worked out by M. Tamminen in 1985 [11]: "Let's assume that all keys are fractions from range [0..1). First, we scatter all N records to $N/8$ baskets, putting key K into basket $KN/8$ (...)".

While looking through described algorithms we can see considerable differences concerning mainly merging of sequence element. Pbit merges the lists only when the lists have been sorted. " We group list elements till the last key. When the last key is looked through, all we have to do is to merge sorted fragments of the list. Every few, maximum M/K, recurrent calls sorted fragment of the list is (recursion) merged." Recursion and the part of code responsible for merging sorted fragments of the list make Pbit very efficient and *innovated* algorithm.

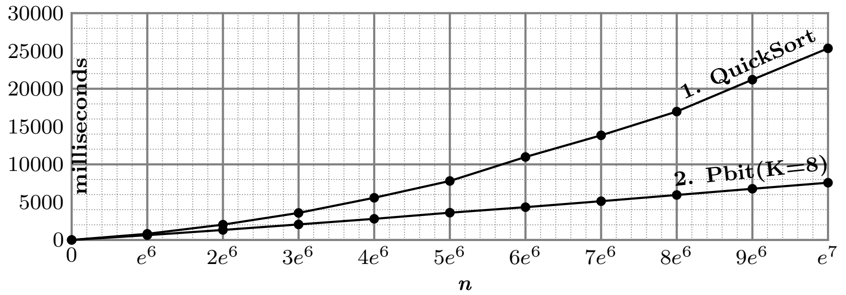
```
if(M) for(int i=0; i<Ω; i++) P=Pbit(tab[i], M, P);
else for(int i=0; i<Ω; i++) P=merge(tab[i], P);
```

All algorithms sorting with "bit key" are very *alike*. Very similar are also all algorithms which sort by comparing elements. However, *comparison implementation* makes QuickSort faster than In-

sertionSort. Just like sorting with "bit key" implementation makes Pbit the most efficient algorithm which sorts lists.

12 Tables versus list

What are the most important differences between a table and list? Firstly, tables have fixed size whereas list size depends on the size of data stored there, enlarged by the size of storage area necessary to keeps pointers for its every element. Secondly, to change the order of list elements, it is enough to change whole a few pointers, which is far less expensive than moving whole portion of memory in case of change of table elements sequence. Thirdly, it's possible to add new elements to the list and remove elements without changing the position of others. There differences help draw the conclusion that if changes will take place into data set very often, especially when the number of set members is not predictable, then it is good to create a list in order to store data set.
Nobody has compared algorithms sorting tables and lists. Why? Because table elements can be freely looked through. Nodes in singly-linked lists have pointer only for the next node, which makes looking through the lists difficult. Moreover, it's faster to refer to elements of the table than nodes of the lists. So far the fastest list sorting algorithm was QuickerSort, much slower than linear-logarithmic tables sorting algorithms. It is good to choose tables when we want to have quick access to data and when data were to be sorted. For the first time in the history of sorting there this can compete with the fastest algorithms tables sorting!



n	e^6	$2e^6$	$3e^6$	$4e^6$	$5e^6$	$6e^6$	$7e^6$	$8e^6$	$9e^6$	e^7
1.	811	2023	3575	5578	7801	10976	13840	16984	21200	25346
2.	631	1322	2053	2794	3605	4336	5127	5948	6770	7571

For this test I used algorithm quicksort from standard library C

13 Double-linked lists sorting

We use double-linked lists very often. How to adopt Pbit for sorting such lists?
In case of double-linked lists sorting we do not change Pbit's code. We only add

repair function "reverse connection" of list elements (nodes). After (double-linked) list sorting using Pbit, pointers showing the previous nodes are incorrect. Function which changes "reverse connection" should receive root of the list as parameter. Next, in turns going through its elements it should set up pointer 'back' in such a way as to show the element right behind.

14 Floating-point numbers sorting

Floating-point numbers in digital computer written in the following form[2]

$$l = m * 2^c,\ l \text{ - number},\ m \text{ - mantissa},\ c \text{ - characteristic}$$

In principle, characteristic is more important than mantissa for the process of sorting. This is why we can sort the number in relation to mantissa first, and then once more in relation to its characteristic. Such sorting is correct because Pbit is stable. You should remember about conversion, because logical shift operator does not work with floating-point numbers!

Floating-point formats (standard ANSI IEEE 754):
— simple, single precision (SINGLE): $m = 23 + 1, c = 8$
— extended, single precision (SINGLE, EXTENDED): $m >= 32, c >= 11$
— simple, double precision (DOUBLE): $m = 52 + 1, c = 11$
— extend, double precision (DOUBLE, EXTENDED): $m >= 64, c >= 15$

15 Portability

Byte order[3] It's obvious that all modern machines have 8-bit bytes. But in different machines there are different representations of object greater than one byte. For example, integers type 'short' (language C), which typical have two bytes, may be stored in memory in two ways: their less significant byte will be at smaller address (less significant byte first, so little-endian order of bytes) or inversely - at greater address (more significant byte first, so big-endian order of bytes) than more significant byte. Although machines in both cases treat memory as sequence of words in the some order, they interpret the byte order within the words differently. This is why it's important to change the lines of code which is responsible for looking through bits by Pbit (the version of algorithm described in documentation was implemented by machine of little-endian type).

[2] The problem of binary arrangements of float-point numbers was well described by Daniel W. Lewis [12]

[3] The problem of code portability was well described by Kernighan and Pike [13].

```
#include<stdio.h>
int main(){
unsigned long x=0x11223344UL;
unsigned int size=sizeof(x);
unsigned char *p=(unsigned char *)&x;
while(size--) printf("%x, ",*p++);
return 0;
}
```

To check how our machine implements the byte order within the word we can use a simple program (standard PC is machine of little-endian type)

The result of this program in machine with decreasing order of bytes (Motorola 68K): 11, 22, 33, 44,
In case of machine with ascending order of bytes (80x86 Intel Architecture): 44, 33, 22, 11,
Machine type PDP-11 (old-fashioned 16-bit machine) will give the result: 22, 11, 44, 33,

Arithmetic or logical shift Shift of value with number sign to the right by means of operator '`>>`' can be treated as arithmetic shift (the copy of bit sign is copied during bit shift) or logical (zero will be placed in released bits during the shift). Fortunately, this problem does not concern Pbit, despite the fact that I use shift operator. It happens that way because operation of logical '`AND`' comes after the operation of bit shift, which unifies the manner of bit shift.

Gratefulness

I do extend thanks to Marek Gajecki – scientist who has been teaching me everything about algorithms. I want to special thanks to Tomasz Jurczyk for effective implementation QuickerSort and MergeSort.

References

[1] Kenneth A. Reek: *Pointers on C*, Addison Wesley Longman, Inc 1998

[2] E. J Issac, R. C. Singelton: *Sorting by address calculation*, Journal of the ACM 13, 169-174, 1956.

[3] Harold H. Seward: *Master's thesis*, MIT Digital Computer Laboratory Report R-232, 1954.

[4] J. Katajainen, T. Pasanen, J. Teuhola: *Practical in-place mergesort*, Nordic Journal of Computing 3, 27 - 40, 1996.

[5] R.S. Scowen: *Algorithm 271: QuickerSort*, Communications of ACM 8, 669-670, 1965.

[6] Alfred V. Aho, Jeffrey D. Ullman: *Foundation of Computer Science in C*, W. H. Freeman and CO, 1995

[7] Wegner, L. M. : *Quicksort for Equal Keys*, IEEE Transactions on Computers C-34 4, 362-367, April 1985.

[8] Donald E. Knuth: *The Art of Computer Programming* Vol. 3, Addison Wesley Longman, Inc. 1998

[9] M. D. MacLaren: *Internal sorting by radix plus shifting*, Journal of the ACM 13, 404-411, 1966.

[10] W. Dobosiewicz: *Sorting by distributive partitioning*, Inf. Proc. Letters 7, 1-6, 1978.

[11] Markku Tamminen: *Two Levels are as Good as Any*, Journal of Algorithms 6, 138-144, 1985.

[12] Daniel W. Lewis: *Where C and Assembly Meet*, Prentice Hall 2002

[13] Brian W. Kernighan, Rob Pike: *Practice of Programming*, Addison Wesley Longman, Inc. 1999