PrimeLister v2.9: Hybrid-Optimized Prime Generation via Segmented Marking

Tobias Jung
info@scyrus.de

January 30, 2026

Abstract

This paper documents *PrimeLister v2.9*, a C++ implementation that generates all primes up to a given limit N. The focus is on a hybrid, segmented marking approach (cache-friendly segmentation, OpenMP parallelization) and on providing the full source code for reproducibility.

1 Introduction

Prime tables are used in number theory, cryptography, and performance benchmarking. This document describes the approach implemented in `Version2.9.txt` and provides reproducibility notes.

2 Algorithmic Approach (Overview)

The code operates on segments of an index space and marks composite candidates using multiple formulas/steps. The segment size is chosen adaptively to improve cache locality and load balancing (with OpenMP).

2.1 Index Mapping (Index to Candidate)

Internally, the algorithm does not represent all integers. Instead it enumerates the $6k \pm 1$ candidates only. In the C++ implementation this is done via a 0-based index m and the mapping

$$p(m) = \begin{cases} 3m+5, & m \text{ even} \\ 3m+4, & m \text{ odd} \end{cases}$$

which yields the ordered sequence $5, 7, 11, 13, 17, 19, \ldots$ (and finally 2 and 3 are prepended explicitly). Marking is performed in this index space: a marked index means the corresponding candidate $p(m)$ is composite.

2.2 Why the Offsets Appear (and a Correctness Sketch)

The three start indices used in the ImageJ prototype (and mirrored in the C++ version) were found empirically by the author (manual derivation with printed candidate lists and pen-and-paper). They are closed-form descriptions of where products of already-generated candidates land in the index space. Concretely, the offsets $7i + 10$, $3i^2 + 8i + 4$, and $3i^2 + 10i + 7$ are the first index positions (for alternating parity classes of i) at which the elimination sequences should start so that stepping by $2 \cdot A[i]$ hits exactly those indices whose mapped values are multiples of the current candidate. Because every prime > 3 is of the form $6k \pm 1$, every composite number that survives the trivial pre-marking (multiples of 5 and 7 in the C++ code) still has a smallest prime factor of the same form. Thus, for each composite candidate there exists an index i corresponding to its smallest factor, and one of the three marking schemes reaches the index of that composite (with the square-handling step covering the remaining square cases), ensuring that all composites in the candidate set are eventually marked.

2.3 Core Idea

The original prototype was developed as an ImageJ macro; this version is often easier to read for understanding the core idea. The approach was then ported to C++ with the help of Gemini and optimized for performance (including OpenMP parallelization and segmented bit marking).

Both variants (C++ and the ImageJ macro) share the same basic construction: only candidates of the form $6k \pm 1$ are considered (2 and 3 are added separately). Composite candidates are then removed/marked in three systematic steps:

1. Remove certain multiples from the subsequence $E_0, E_2, E_4, \dots$ (macro: start index $7i + 10$ with step size $2 \cdot A[i]$).
2. Remove certain multiples from the subsequence $E_1, E_3, E_5, \dots$ (macro: start index $3i^2 + 8i + 4$ with step size $2 \cdot A[i]$).
3. Remove the remaining squares (macro: index $3i^2 + 10i + 7$) and their subsequent elements.

The C++ implementation translates these steps into cache-friendly segmented bit marking and parallelizes segment processing using OpenMP.

3 Implementation Details

3.1 Parallelization

If OpenMP is available, segments are processed in parallel and results are collected per segment.

3.2 File Output

Results are written to `PrimeList.txt` as a table with `cols` columns.

4 Experimental Setup and Results

Benchmark values in Table 1 are rounded means from 5 program runs per configuration. The program was restarted between runs, and no other programs were running in the background. Across repeated runs on the same machine, the program suggested the same number of threads.

The reported throughput in Table 1 (*primes/s*) refers to the core prime computation only (as measured and printed by the program), excluding the file output step (writing `PrimeList.txt`). Existing benchmarks (multiple systems) are included as slides in `Benchmark 002.pptx`.

4.1 Benchmark Systems

- **Slow System:** Intel(R) Core(TM) i7-9750H CPU, 2.60GHz, 6 cores, 12 threads; RAM: 16 GB; OS: Windows 11 (64-bit).
- **Medium System:** Intel(R) Core(TM) Ultra 7 155U, 1.70 GHz, 12 cores, 14 threads; RAM: 64 GB; OS: Windows 11 (64-bit).
- **Fast System:** Intel(R) Xeon(R) w7-3455, 2.50 GHz, 24 cores, 48 threads; RAM: 512 GB; OS: Windows 11 (64-bit).

N	Slow System (primes/s)	Medium System (primes/s)	Fast System (primes/s)
1,000,000	35,279,999	22,620,598	16,016,731
100,000,000	75,872,411	113,673,142	68,046,243
1,000,000,000	61,019,767	121,854,039	145,904,389
2,000,000,000	46,757,637	103,569,420	120,028,926
5,000,000,000	21,305,698	62,130,239	79,255,786
10,000,000,000	11,990,755	35,989,927	72,354,973

Table 1: Benchmarks from `Benchmark fÃ¼r ArchiveX.csv` (PrimeLister v2.9, "Calculate primes below"). Values are in *primes/s*.

5 Discussion and Related Work

At a high level, PrimeLister follows the same core ideas as established high-performance prime generators: a segmented sieve for cache efficiency, pre-sieving of small prime factors, and parallelization across segments. These ingredients are well-known performance techniques for prime sieving.[1, 2, 3]

The distinctive aspect of the present work is the closed-form index arithmetic used in the ImageJ prototype (three start indices and step sizes) and its translation into an indexed marking scheme in the C++ implementation. A targeted literature search for the exact offset formulas used here did not reveal an earlier publication that matches these specific expressions; however, functionally related approaches (segmented sieves and wheel factorization / pre-sieving) are widely used.[1, 2]

5.1 Prior Dissemination

The underlying idea was presented publicly in a three-part YouTube video series by the author (Part 1/2: video IDs `58EMZtSLT9Q` and `N053-sZHmj4`; PrimeLister 2.9 update: video ID `dpT2nV7t1p0`).

6 Limitations and Future Work

The current benchmark table reports throughput in primes/s but does not yet record the exact compiler flags (e.g., optimization level, OpenMP settings) or the run protocol (number of runs, warm-up, and aggregation statistic). Future work includes adding a standardized benchmarking harness (fixed number of repetitions and reporting median/variance), documenting build flags precisely, and extending the evaluation to additional platforms/compilers.

7 Reproducibility

- Source code: see Appendix A (C++) and Appendix B (ImageJ macro).
- Repository: GitHub `StrangestThings/PrimeLister-2.9-Multicore`.
- Build environment: Microsoft Visual Studio 2026 (MSVC toolset 14.44.35207, Release x64).
- Key flags (excerpt): /O2 /Ot /GL /arch:AVX2 /std:c++20 /permissive- /MD /openmp:llvm /fp:precise; linker: /LTCG /OPT:REF /OPT:ICF.
- Full compiler and linker command lines are provided below.
- Benchmark slides: `Benchmark 002.pptx` (measurements on multiple systems).
- Inputs: limit N and the number of columns `cols`.
- Output: `PrimeList.txt`.
- License: CC0 1.0 Universal (public domain dedication).

7.1 MSVC Build Log (Command Lines)

Compiler (CL.exe):

```
C:\Program Files\Microsoft Visual Studio\18\Community\VC\Tools\MSVC\14.44.35207\bin\HostX64\x64\CL.
    exe /c /I"C:\Users\user\source\repos\PrimeGeneratorMulticore\vcpkg\installed\x64-windows\
    include" /Zi /nologo /W3 /WX- /diagnostics:column /sdl /MP /O2 /Oi /Ot /GL /D NDEBUG /D
    _CONSOLE /D _UNICODE /D UNICODE /Gm- /EHsc /MD /GS /Gy /arch:AVX2 /fp:precise /Zc:wchar_t /Zc:
    forScope /Zc:inline /openmp /std:c++20 /permissive- /Fo"x64\Release\\" /Fd"x64\Release\vc143.
    pdb" /external:W3 /Gd /TP /FC /errorReport:prompt /openmp:llvm
```

Linker (link.exe):

```
C:\Program Files\Microsoft Visual Studio\18\Community\VC\Tools\MSVC\14.44.35207\bin\HostX64\x64\
    link.exe /ERRORREPORT:PROMPT /OUT:"C:\Users\user\source\repos\PrimeGeneratorMulticore\x64\
    Release\PrimeGeneratorMulticore.exe" /NOLOGO /LIBPATH:"C:\Users\user\source\repos\
    PrimeGeneratorMulticore\vcpkg\installed\x64-windows\lib" /LIBPATH:"C:\Users\user\source\repos\
    PrimeGeneratorMulticore\vcpkg\installed\x64-windows\lib\manual-link" kernel32.lib user32.lib
    gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib
    odbc32.lib odbccp32.lib "C:\Users\user\source\repos\PrimeGeneratorMulticore\vcpkg\installed\x64
    -windows\lib\*.lib" /MANIFEST /MANIFESTUAC:"level='asInvoker' uiAccess='false'" /manifest:embed
     /DEBUG /PDB:"C:\Users\user\source\repos\PrimeGeneratorMulticore\x64\Release\
    PrimeGeneratorMulticore.pdb" /SUBSYSTEM:CONSOLE /OPT:REF /OPT:ICF /LTCG /LTCGOUT:"x64\Release\
    PrimeGeneratorMulticore.iobj" /TLBID:1 /DYNAMICBASE /NXCOMPAT /IMPLIB:"C:\Users\user\source\
    repos\PrimeGeneratorMulticore\x64\Release\PrimeGeneratorMulticore.lib" /MACHINE:X64 x64\Release
    \PrimeGeneratorMulticore.obj
```

References

[1] GeeksforGeeks. *Segmented Sieve.*

[2] Kim Walisch. *primesieve* (documentation / manual).

[3] Stephan Brumme. *Parallel Prime Sieve.*

8 License and Attribution

The author dedicates the source code and benchmark data in this project to the public domain under the CC0 1.0 Universal dedication.

The underlying formulas and the original implementation were developed by Tobias Jung in ImageJ; the C++ port and parallelization were produced with the assistance of Gemini-3.

A Source Code

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <cmath>
#include <cstdint>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <charconv>
#include <cstring>
#include <thread>

// ---- OpenMP Setup ----
#ifdef _OPENMP
#include <omp.h>
#else
#include <chrono>
inline double omp_get_wtime() {
    static const auto t0 = std::chrono::steady_clock::now();
    return std::chrono::duration<double>(std::chrono::steady_clock::now() - t0).count();
}
inline int omp_get_num_procs() { return (int)std::thread::hardware_concurrency(); }
#endif

#if defined(_WIN32)
#ifndef NOMINMAX
#define NOMINMAX
#endif
#include <windows.h>
#include <intrin.h>
#endif

using u64 = unsigned long long;

// ---------- Optimierte Helfer ----------
static inline u64 ceil_div_u64(u64 a, u64 b) {
    return (a + b - 1) / b;
}

static inline unsigned ctz64(uint64_t x) {
```

```
#if defined(_MSC_VER)
    unsigned long idx;
    _BitScanForward64(&idx, x);
    return (unsigned)idx;
#else
    return (unsigned)__builtin_ctzll(x);
#endif
}

// ---------- Datei-Ausgabe ----------
static inline void buf_append_u64(std::string& buf, u64 x) {
    char tmp[32];
    auto res = std::to_chars(tmp, tmp + 32, x);
    buf.append(tmp, (size_t)(res.ptr - tmp));
}

static void write_prime_table(const std::vector<u64>& P, u64 cols) {
    std::ofstream out("PrimeList.txt", std::ios::out | std::ios::binary);
    if (!out) return;

    std::string buf;
    buf.reserve(16 * 1024 * 1024);

    u64 count = 0;
    for (size_t i = 0; i < P.size(); ++i) {
        if (count == 0) {
            buf_append_u64(buf, (u64)(i + 1));
            buf.push_back('-');
            buf_append_u64(buf, (u64)(std::min)((size_t)(i + cols), P.size
                ()));
            buf.push_back('\t');
        }
        buf_append_u64(buf, P[i]);
        buf.push_back('\t');
        if (++count == cols) {
            buf.push_back('\n');
            count = 0;
        }
        if (buf.size() >= 15 * 1024 * 1024) {
            out.write(buf.data(), (std::streamsize)buf.size());
            buf.clear();
        }
    }
    if (count != 0) buf.push_back('\n');
    out.write(buf.data(), (std::streamsize)buf.size());
}

// ---------- Kern-Algorithmus v2.9 (HYBRID) ----------
static std::vector<u64> primes_formeln(u64 N, u64 cols, bool silent) {
    u64 bereich = (N + 2ull) / 3ull + (15ull * (u64)(std::max)((size_t)1,
        (size_t)cols));
    if (bereich == 0) return {};

    int T = 1;
```

```
#ifdef _OPENMP
    T = omp_get_max_threads();
#endif

    // ADAPTIVE: mehr Segmente fuer besseres Load-Balancing bei grossen N
    u64 target_segments = (N > 1000000000ull) ? ((u64)T * 6ull) : ((u64)T * 4ull);
    u64 SEG_IDX = (bereich + target_segments - 1) / target_segments;

    // Cache-optimierte Grenzen
    const u64 MIN_SEG = 1ull << 23;  // 1 MB (kleiner fuer kleine N)
    const u64 MAX_SEG = 1ull << 27;  // 16 MB
    if (SEG_IDX < MIN_SEG) SEG_IDX = MIN_SEG;
    if (SEG_IDX > MAX_SEG) SEG_IDX = MAX_SEG;
    if (SEG_IDX > bereich) SEG_IDX = bereich;

    u64 nsegs = (bereich + SEG_IDX - 1) / SEG_IDX;
    const long long M = (long long)bereich;

    long long end1 = (M - 11) / 7;
    if (end1 < 0) end1 = -1;

    long long end2 = (M <= 4) ? -1 : (long long)std::floor((-8.0L + std::sqrt(16.0L + 12.0L * (long double)M)) / 6.0L);
    long long end3 = (M <= 7) ? -1 : (long long)std::floor((-10.0L + std::sqrt(16.0L + 12.0L * (long double)M)) / 6.0L);

    std::vector<std::vector<u64>> buckets((size_t)nsegs);

    double density = 1.0 / std::log((double)N + 1.0);
    size_t estimated_per_seg = (size_t)(SEG_IDX * 3.0 * density * 1.15);
    for (auto& b : buckets) {
        b.reserve(estimated_per_seg);
    }

#pragma omp parallel
    {
        std::vector<uint64_t> mark_words;
        mark_words.reserve((size_t)((SEG_IDX + 63ull) >> 6));

#pragma omp for schedule(guided, 1)
        for (long long s = 0; s < (long long)nsegs; ++s) {
            u64 base = (u64)s * SEG_IDX;
            u64 end = (u64)(std::min)((size_t)bereich, (size_t)(base + SEG_IDX));
            size_t seg_len = (size_t)(end - base);
            size_t nwords = (seg_len + 63u) >> 6;

            mark_words.resize(nwords);
            std::memset(mark_words.data(), 0, nwords * sizeof(uint64_t));

            long long Mend = (long long)end;
            long long e1seg = (long long)(std::min)((size_t)end1, (size_t)((Mend - 11) / 7));
```

```
            long long e2seg = (Mend <= 4) ? -1 : (long long)(std::min)((size_t)end2, (size_t)std::floor((-8.0L + std::sqrt(16.0L + 12.0L * (long double)Mend)) / 6.0L));
            long long e3seg = (Mend <= 7) ? -1 : (long long)(std::min)((size_t)end3, (size_t)std::floor((-10.0L + std::sqrt(16.0L + 12.0L * (long double)Mend)) / 6.0L));

#define SETBIT(idx) mark_words[(idx) >> 6] |= (1ull << ((idx) & 63))

            u64 r10 = base % 10ull;
            u64 r14 = base % 14ull;

            u64 i1 = (r10 == 0 ? base : base + (10 - r10));
            if (i1 == 0 && base == 0) i1 = 10;
            for (u64 i = i1; i < end; i += 10) SETBIT((size_t)(i - base));

            for (u64 i = base + (r10 <= 7 ? 7 - r10 : 17 - r10); i < end; i += 10)
                SETBIT((size_t)(i - base));

            for (u64 i = base + (r14 <= 10 ? 10 - r14 : 24 - r14); i < end; i += 14)
                SETBIT((size_t)(i - base));

            u64 i14 = base + (r14 <= 1 ? 1 - r14 : 15 - r14);
            if (i14 == 1 && base == 0) i14 = 15;
            for (u64 i = i14; i < end; i += 14) SETBIT((size_t)(i - base));

            // Formel 1: Simpler Loop (kein Unrolling bei grossem Overhead)
            for (long long i = 0; i <= e1seg; i += 2) {
                u64 iu = (u64)i;
                u64 step = 6ull * iu + 10ull, j0 = 7ull * iu + 10ull;
                if (j0 < base) j0 += ceil_div_u64(base - j0, step) * step;
                for (u64 j = j0; j < end; j += step) SETBIT((size_t)(j - base));
            }

            // Formel 2
            for (long long i = 1; i <= e2seg; i += 2) {
                u64 iu = (u64)i;
                u64 step = 6ull * iu + 8ull;
                u64 j0 = 3ull * iu * iu + 8ull * iu + 4ull;
                if (j0 < base) j0 += ceil_div_u64(base - j0, step) * step;
                for (u64 j = j0; j < end; j += step) SETBIT((size_t)(j - base));
            }

            // Formel 3: Branch-Optimierung
            for (long long i = 0; i <= e3seg; i += 2) {
                u64 iu = (u64)i;
                u64 step = 6ull * iu + 10ull;
                u64 idx = 3ull * iu * iu + 10ull * iu + 7ull;
```

```
                if (idx < base) {
                    u64 j0 = idx + ceil_div_u64(base - idx, step) * step;
                    for (u64 j = j0; j < end; j += step) SETBIT((size_t)(j
                        - base));
                }
                else if (idx < end) {
                    SETBIT((size_t)(idx - base));
                    for (u64 j = idx + step; j < end; j += step) SETBIT((
                        size_t)(j - base));
                }
            }

#undef SETBIT

            std::vector<u64>& local = buckets[(size_t)s];

            for (size_t wi = 0; wi < nwords; ++wi) {
                uint64_t inv = ~mark_words[wi];
                while (inv) {
                    unsigned b = ctz64(inv);
                    size_t off = (wi << 6) + b;
                    if (off >= seg_len) break;
                    u64 idx_v = base + off;
                    u64 p = ((idx_v & 1ull) == 0ull) ? (3ull * idx_v + 5
                        ull) : (3ull * idx_v + 4ull);
                    if (p <= N) local.push_back(p);
                    inv &= (inv - 1ull);
                }
            }
        }
    }

    std::vector<u64> primes;
    size_t total_p = 1 + (N >= 3 ? 1 : 0);
    for (const auto& b : buckets) total_p += b.size();
    primes.reserve(total_p);

    primes.push_back(2);
    if (N >= 3) primes.push_back(3);

    for (const auto& b : buckets) {
        primes.insert(primes.end(), b.begin(), b.end());
    }

    return primes;
}

// ---------- MAIN ----------
int main() {
    std::ios::sync_with_stdio(false); std::cin.tie(NULL);

#if defined(_WIN32)
    SetPriorityClass(GetCurrentProcess(), HIGH_PRIORITY_CLASS);
```

```cpp
#endif

    std::cout << "**************************************************\n";
    std::cout << "*  PrimeLister v2.9 Hybrid-Optimiert (2026)      *\n";
    std::cout << "**************************************************\n\n";

    int max_t = omp_get_num_procs();
    int best_t = 1; double min_t = 1e30;
    u64 calib_N = 1000000000;

    std::cout << "[Step 1] Kalibriere Hardware mit N = 1.000.000.000...\n";
    std::vector<int> test_configs = { 1, max_t / 4, max_t / 2, (3 * max_t) / 4, max_t };

    for (int t : test_configs) {
        if (t < 1 || t > max_t) continue;
#ifdef _OPENMP
        omp_set_num_threads(t);
#endif
        double start = omp_get_wtime();
        auto d = primes_formeln(calib_N, 1, true);
        double end = omp_get_wtime() - start;
        std::cout << "  - Threads " << std::setw(2) << t << ": "
            << std::fixed << std::setprecision(4) << end << "s\n";
        if (end < min_t) { min_t = end; best_t = t; }
    }
#ifdef _OPENMP
    omp_set_num_threads(best_t);
#endif
    std::cout << ">> Optimal: " << best_t << " Threads ("
        << std::fixed << std::setprecision(2) << (min_t * 1000) << " ms).\n\n";

    u64 N; u64 cols;
    std::cout << "[Step 2] Parameter eingeben\nLimit N: "; std::cin >> N;
    std::cout << "Spalten: "; std::cin >> cols;

    std::cout << "\n[Step 3] Hauptberechnung laeuft...\n";
    double t0 = omp_get_wtime();
    auto P = primes_formeln(N, cols, false);
    double t1 = omp_get_wtime();

    std::cout << "\nFirst 100 primes:\n";
    size_t limit_first = (P.size() < 100) ? P.size() : 100;
    for (size_t i = 0; i < limit_first; ++i) std::cout << P[i] << ' ';

    std::cout << "\n\nLast 10 primes under limit:\n";
    size_t start_last = (P.size() > 10) ? (P.size() - 10) : 0;
    for (size_t i = start_last; i < P.size(); ++i) std::cout << P[i] << ' ';

    std::cout << "\n\nOverall primes found: " << P.size() << "\n";
```

```
    std::cout << "Overall calculation time [s]: " << std::fixed << std::
        setprecision(6) << (t1 - t0) << "\n";

    if (t1 - t0 > 0) {
        double rate = (double)P.size() / (t1 - t0);
        std::cout << "~ " << (u64)rate << " primes/s\n";
    }

    std::cout << "\nWriting PrimeList.txt ...\n";
    double tw0 = omp_get_wtime();
    write_prime_table(P, cols);
    double tw1 = omp_get_wtime();
    std::cout << "Write time: " << std::fixed << std::setprecision(3) << (
        tw1 - tw0) << "s\n";

    std::cout << "\nDone. Press Enter to exit...";
    std::cin.ignore(1000, '\n'); std::cin.get();
    return 0;
}
```

B Source Code (ImageJ)

```
time=getTime();
print("\\Clear");

// 16GB-Ram: Not more than 100 Mio.

// Calculates about 0.7 sec per Million

bereich=100000;

A=newArray(bereich);


print("Array was created: " + IJ.freeMemory());

// Fill Array A with prime candidates
k=1;
for (i = 1; i < A.length; i=i+2) {A[i-1]=6*k-1; A[i]=6*k+1; k++;}
// ******************************************************

print("Array filled: " + IJ.freeMemory());

print(" Delete all the multiples of the single elements (E0, E2, E4...)");

for(i = 0; (7*i+10) < A.length; i=i+2){
while (A[i]<1){i=i+2;}
step=2*A[i];
for (j = 7*i+10; j < A.length; j=j+step) {A[j]=0;}
}

print(" Delete all the multiples of the single elements (E1, E3, E5...)");
```

```

for(i = 1; (3*i*i+8*i+4) < A.length; i=i+2){
while (A[i]<1){i=i+2;}
step=2*A[i];
for (j = 3*i*i+8*i+4; j < A.length; j=j+step) {A[j]=0;}
}

print(" Now delete all the remaining squares of the single elements (E0,
    E2, E4...)");

for (i = 0; (3*i*i+10*i+7) < A.length; i=i+2) {
if(A[3*i*i+10*i+7]>0){step=2*sqrt(A[3*i*i+10*i+7]); A[3*i*i+10*i+7]=0; for
     (j =(3*i*i+10*i+7); j < A.length; j=j+step) {A[j]=0;}}
}

// Delete Zeros

A=Array.deleteValue(A, 0);

// Attach 2 and 3
A=Array.concat(A,2);
A=Array.concat(A,3);

A=Array.sort(A);

time2=getTime();
zeit=(time2-time)/1000;

print("Range: " + bereich);
print("Primes identified: " + A.length);
print("Calculation time  [sec]: " + zeit);
print("Primes per second: " + A.length/zeit);


print("**************************************************");

results = getInfo("log");
results=replace(results,".",",");
print("\\Clear");
print(results);

// Output of 100 primes per line with the according counter


spalten=100;

for (i = 0; i < A.length-spalten; i=i+spalten) {
line=toString((1+i) + "-" + (i+spalten) + ":" + "\t");
for (s = i; s < i+spalten; s++) {line=line+A[s]+"\t";}
print(line);
}
```