Samenvatting algoritmen

Aaron Mousavi (aanvullingen Jelle De Bock)

March 22, 2017

1 Asymptotische notaties

De naam "asymptotische notatie" is gekozen omdat deze notaties enkel geldig zijn voor grote waarden van n.

- $\Theta(n)$: de functie is langs boven en onder begrensd met de constanten c_1 en c_2, daarom ligt de uitvoeringstijd tussen $c_1 \cdot f(n)$ en $c_2 \cdot f(n)$ ligt.
- $O(n)$: is maximum uitvoeringstijd, het zal nooit slechter zijn dat dit. Er wordt geprobeerd om de laagste, bewezen waarde op te schrijven. We kunnen ook zeggen dat het worst case scenario $\Theta(n)$ is, maar de big O is een sterkere notatie.
- $\Omega(n)$: de ondergrens, het algoritme heeft een uitvoeringstijd van minstens n.

2 Selection sort

Kenmerken

- Niet stabiel
- Rangschikt ter plaatste
- $\Theta(n^2)$
- Ofwel het kleinste zoeken en vooraan zetten, of het grootste en achteraan zetten.

Algoritme

```
1  template <typename T>
2  void SelectionSort<T>::operator()(vector<T> &vec) const {
3      for(int i=vec.size()-1; i>=0; i--){
4          int maxIndex=zoekGrootste(vec, i);
5          swap(vec[i], vec[maxIndex]);
6      }
7  }
8
9  template <typename T>
10 int SelectionSort<T>::zoekGrootste(const vector<T> & vec, int eindIndex)
       const {
11     int maxIndex = 0;
12     for(int i=1; i<=eindIndex; i++){
13         if(vec[i] > vec[maxIndex]){
14             maxIndex = i;
15         }
16     }
17     return maxIndex;
18 }
```

Opmerkingen

- Lidfuncties zijn const, en de vector in de zoekGrootste ook!

3 Insertion sort

Kenmerken

- Stabiel
- Rangschikt ter plaatste
- $O(n^2)$
- $\Omega(n)$

Algoritme

```
1  template <typename T>
2  void InsertionSort<T>::operator()(vector<T> &vec) const {
3      for(int i=1; i<vec.size(); i++){
4          T help;
5          //Voeg element i toe aan het gesorteerde deel
6          //(de elementen voor i staan in volgorde)
7          help = std::move(v[i]);
8          int j = i-1;
9          while(j>=0 && help<v[j]){
10             //help is kleiner dan het voorgaande element dus
11             //laat element op j plaats maken (schuif op)
12             v[j+1] = std::move(v[j]);
13             j--;
14         }
15         //na de voorgaande lus hebben we de positie van help gevonden
16         //merk op: j+1 we zijn namelijk 1tje te ver gegaan in de while lus
17         v[j+1] = std::move(help);
18     }
19 }
```

Opmerkingen

- Const niet vergeten.
- Moves!
- i begint bij 1, regel 8 zeker niet vergeten en op het laatste is het ook j+1!

4 Shell sort

Kenmerken

- Niet stabiel
- Rangschikt ter plaatste
- $O(n^2)$
- $\Omega(n \cdot log(n))$

Algoritme

```
1  //Sedgewick's increments, used in the shellsort algorithm
2  vector<int> INCREMENTS = {  1391376, 463792, 198768, 86961, 33936,
3                              13776, 4592, 1968, 861, 336,
4                              112, 48, 21, 7, 3, 1
5  };
6
7  template <typename T>
8  void Shellsort<T>::operator()(vector<T> & v) const {
9      //Ga over alle increments
10     for(int i = 0 ; i < INCREMENTS.size() ; i++){
11         int increment = INCREMENTS[i];
12         //Controleer of increment niet te groot is voor onze array
13         if(increment < v.size()){
14             //Ga k-sorteren voor alle elementen startende van het increment
15             for(int j = increment ; j < v.size() ; j++){
16                 T temp = v[j];
17                 //Insertion sort stap, plaats j-de element op zijn juiste
                       plek, telkens increment-stapjes verder springend
18                 int k = j-increment;
19                 while(k>=0 && temp < v[k]){
20                     v[k+increment] = v[k];
21                     k-=increment;
22                 }
23                 v[k+increment]=std::move(temp);
24
25                 /* Evenwaardig getest alternatief
26                     int k = j;
27
28                     while(k-increment>=0 && v[k-increment]>temp){
29                         v[k]=std::move(v[k-increment]);
30                         k-=increment;
31                     }
32                     v[k]=std::move(temp);
33                 */
34             }
35         }
36     }
37 }
```

Opmerkingen

- Bij de laatste regel weer de +incr niet vergeten!

5 Heap sort

Kernmerken

- Niet Stabiel
- Ter plaatse
- $\Theta(n \cdot log(n))$

Algoritme heap sort

```
template <typename T>
void HeapSort<T>::operator()(vector<T> &vec) const {
    Heap<T> heap(vec, vec.size());
    heap.buildHeap();
    for (size_t i = vec.size()-1; i > 0; i--) { // opgepast i > 0!!!
        // zet wortel (= grootste achteraan) en zet leaf bovenaan
        T wortel = move(*heap.geefWortel());
        heap.vervangWortel(vec[i]);
        vec[i] = move(wortel);
        heap.percolateDown(0, i);
    }
}
```

Algoritmes heap

```
template<class T>
void Heap<T>::buildHeap(){
    // dit moet enkel voor de bovenste helft
    for (int i = vec.size()/2; i >= 0; i--) {
        percolateDown(i, vec.size());
    }
}

template<class T>
const T* Heap<T>::geefWortel() const {
    return &vec[0];
}

template<class T>
int Heap<T>::getGrootsteKind(int parentIndex, int length){
    int kind = parentIndex * 2 + 1; // moet bestaan, dus niet oproepen op
        leaves! Plus 1 want zero based.
    if(kind + 1 < length){
        if(vec[kind+1] > vec[kind]){
            kind+=1;
        }
    }
    return kind;
}

template<class T>
void Heap<T>::percolateDown(size_t parentIndex, size_t length){
    if(parentIndex * 2 + 1 >= length) {
```

```
28          return; // parent heeft geen kinderen, en kan dus ook niet zakken
29      }
30
31      T parent = move(vec[parentIndex]);
32      size_t grootsteKindIndex = getGrootsteKind(parentIndex, length);
33      // blijf dit doen tot er geen (groter) kind meer is
34      while ((2*parentIndex+1<length) && vec[grootsteKindIndex] > parent) {
35          // parent wordt child
36          vec[parentIndex] = vec[grootsteKindIndex];
37          parentIndex = grootsteKindIndex;
38      }
39      // alle elementen die groter waren dan de originele parent staan er nu
            boven, zet nu de oude parent op de laatste plaats
40      // die waarde hier zit toch al in het niveau erboven
41      vec[parentIndex] = move(parent);
42  }
43
44  template<class T>
45  void Heap<T>::vervangWortel(const T& ele){
46      vec[0] = move(ele);
47  }
```

Opmerkingen

6 Merge sort

Kernmerken

- Stabiel (deze versie toch)
- Niet ter plaatse
- $\Theta(n \cdot log(n))$

Top down Algoritme

```
1  template <typename T>
2  void Mergesort<T>::mergesort(vector<T> &h, vector<T> &v, int l, int r)
       const{
3      if(l+1 < r){
4          int m = l+(r-l)/2;
5          mergesort(v, h, l, m);
6          mergesort(v, h, m, r);
7
8          merge(h, v, l, m , r);
9      }
10 }
11
12 template <typename T>
13 void Mergesort<T>::operator()(vector <T> &v) const {
14     vector<T> other(v);  //de andere tabel, waarin we telkens de rechter
           deeltabel zullen in opslaan
15     mergesort(other,v,0,v.size());
16 }
17
18 template <typename T>
19 void merge(vector<T> &from, vector<T> &to, int l, int m, int r){
20     int i=l;
21     int j=m;
22
23     //je weet exact hoeveel keer je een element gaat plaatsen (dus
           tellertje k)
24     for(int k=i; k<r; k++){
25         if(i<m && (j>=r || from[i] <= from[j])){
26             to[k]=from[i++];
27         }else{
28             to[k]=from[j++];
29         }
30     }
31 }
```

Bottom up Algoritme

```
1  template <typename T>
2  void Mergesort_bu<T>::operator()(vector <T> &v) const {
3      vector<T> temp(v.size());
4      int breedte=1;
5      while(breedte<v.size()){
6          int l=0;
```

```
7           while(l+breedte<v.size()){
8               merge_bu(v, temp, l, breedte);
9               l=l+breedte*2;
10          }
11          breedte*=2;
12      }
13  }
14
15  template <typename T>
16  void merge_bu(vector<T> &vec, vector<T> &temp, int l, int width){
17      int i=l;
18      int j=l+width;
19
20      int m = l+width;
21      int r = l+2*width>vec.size()?vec.size():l+2*width;
22      int k=l;
23
24      for(int s=l; s<r; s++){
25          if(i<m && (j>=r || vec[i] <= vec[j])){
26              temp[s]=vec[i++];
27          }else{
28              temp[s]=vec[j++];
29          }
30      }
31
32      for(int s=l; s<r; s++){
33          vec[s]=temp[s];
34      }
35  }
```

Opmerkingen

- Hoe doe je dit met een hulp array van $n/2$?

7 Quick sort

Single pivot

```
template <typename T>
void Quicksort<T>::operator()(vector <T> &v) const {
    quicksort(v,0, v.size()-1);
}

template <typename T>
void Quicksort<T>::quicksort(vector <T> &v, int l, int r) const {
    //Rangschik deelvector v[l]..[r]
    if(l<r) {
        T pivot = v[l];
        int i=l, j=r;
        while(v[j]>pivot){
            j--;
        }
        while(i<j){
            swap(v[i],v[j]);
            i++;
            while(v[i]<pivot)
                i++;
            j--;
            while(v[j]>pivot)
                j--;
        }
        quicksort(v, l, j);
        quicksort(v, j+1, r);
    }
}
```

Dual Pivot (JDK aka Yaroslavskiy)

```
template <typename T>
void Quicksort_Dual_Pivot<T>::operator()(vector <T> &v) const {
    quicksort_Dual_Pivot(v,0, v.size()-1);
}


template <typename T>
void Quicksort_Dual_Pivot<T>::quicksort_Dual_Pivot(vector <T> &v, int l,
    int r) const {
    if (l < r) {
        //Pivots
        T p1 = v[l];
        T p2 = v[r];

        //Swap the 2 pivots if p1>p2
        if (p1 > p2) {
            v[l] = std::move(p2);
            v[r] = std::move(p1);
            p1 = v[l];
            p2 = v[r];
        }
```

```
21          //m is het tellertje waarmee we door de tabel lopen
22          //k is de grens tussen linker en middelste gedeelte (alle waardes <
                p1)
23          //g is de grens tussen rechter en middelste gedeelte (alle waardes
                >p2)
24          int k = l + 1;
25          int g = r - 1;
26          int m = k;
27
28          while (m <= g) {
29              if (v[m] < p1) {
30                  T temp = std::move(v[k]);
31                  v[k] = std::move(v[m]);
32                  v[m] = std::move(temp);
33                  k++;
34              } else if (v[m] >= p2) {
35                  while (v[g] > p2 && m < g) {
36                      g--;
37                  }
38                  T temp = std::move(v[g]);
39                  v[g] = std::move(v[m]);
40                  v[m] = std::move(temp);
41
42                  g--;
43
44                  if (v[m] < p1) {
45                      T temp = std::move(v[m]);
46                      v[m] = std::move(v[k]);
47                      v[k] = temp;
48                      k++;
49                  }
50              }
51              m++;
52          }
53          //Zet pivots op hun plek
54          k = k - 1;
55          g = g + 1;
56          T temp = std::move(v[k]);
57          v[k] = std::move(v[l]);
58          v[l] = std::move(temp);
59
60          temp = std::move(v[g]);
61          v[g] = std::move(v[r]);
62          v[r] = std::move(temp);
63
64          quicksort_Dual_Pivot(v, l, k - 1);
65          quicksort_Dual_Pivot(v, k + 1, g - 1);
66          quicksort_Dual_Pivot(v, g + 1, r);
67      }
68  }
```

8 Counting sort

Kenmerken

- Stabiel
- Niet ter plaatste
- Sleutels moeten gehele getallen zijn
- Beperkt interval
- $\Theta(n+k)$ met n aantal elementen en k het aantal verschillende sleutels.

Algoritme

```
1  template <typename T>
2  void CountingSort<T>::operator()(vector<T> &vec) const {
3
4      // frequentietabel opstellen
5      vector<int> freq(vec.size(), 0);;
6      for (int i = 0; i < vec.size(); i++) {
7          freq[vec[i]]++; // ervan uitgaande dat vec een getal teruggeeft...
8      }
9
10     // cumulatieve tabel maken
11     for (int i = 1; i < freq.size(); i++) {
12         freq[i] += freq[i-1];
13     }
14
15     // output tabel maken met het resultaat
16     // overloop terug de originele tabel
17     vector<int> output(vec.size());
18     for (int i = 0; i < vec.size(); i++) {
19         output[freq[vec[i]]-1] = move(vec[i]); // -1!!
20         freq[vec[i]]--;
21     }
22
23     // output kopieren naar de originele tabel
24     for (int i = 0; i < vec.size(); i++) {
25         vec[i] = move(output[i]);
26     }
27 }
```

Opmerkingen

9 Radix sort

```
// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;

    // Change count[i] so that count[i] now contains actual
    //  position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

10 Bucket sort

Kenmerken

- Stabiel: door insertion sort
- Niet ter plaatste
- $O(n^2)$: alles in één bucket en insertionsort
- $\Theta(n)$

Algoritme

```
template <typename T>
T BucketSort<T>::grootsteValue(vector<T> const &vec) const {
    // T moet wel zeker getal zijn, of anders een complexere interface
       maken
    T grootste = vec[0];
    for (int i = 0; i < vec.size(); i++) {
        if(vec[i] > grootste) grootste = vec[i];
    }
    return grootste;
}

template <typename T>
void BucketSort<T>::operator()(vector<T> &vec) const {
    // dit houdt alle buckets bij
    vector<vector<T>> buckets(vec.size());
    T grootste = grootsteValue(vec);

    // overloop alle waarden
    for (int i = 0; i < vec.size(); i++) {
        // voorbeeld hash functie: (value*size)/(max value+1)
        int bucket = (vec[i] * vec.size())/(grootste+1);
        buckets[bucket].push_back(move(vec[i]));
    }

    // sorteer alle buckets
    InsertionSort<T> insertionSort;
    for (int i = 0; i < buckets.size(); i++) {
        insertionSort(buckets[i]);
    }

    // leeg alle buckets terug in de array
    int index=0;
    for (int i = 0; i < buckets.size(); i++) {
        for (int j = 0; j < buckets[i].size(); j++) {
            vec[index++] = move(buckets[i][j]);
        }
    }
}
```

Opmerkingen

11 Mogelijks handig

Move implementatie voorbeeld

```
1  // Move constructor.
2  MemoryBlock(MemoryBlock&& other)
3      : _data(nullptr)
4      , _length(0)
5  {
6      std::cout << "In MemoryBlock(MemoryBlock&&). length = "
7                << other._length << ". Moving resource." << std::endl;
8
9      // Copy the data pointer and its length from the
10     // source object.
11     _data = other._data;
12     _length = other._length;
13
14     // Release the data pointer from the source object so that
15     // the destructor does not free the memory multiple times.
16     other._data = nullptr;
17     other._length = 0;
18 }
19
20 // Move assignment operator.
21 MemoryBlock& operator=(MemoryBlock&& other)
22 {
23     std::cout << "In operator=(MemoryBlock&&). length = "
24               << other._length << "." << std::endl;
25
26     if (this != &other)
27     {
28         // Free the existing resource.
29         delete[] _data;
30
31         // Copy the data pointer and its length from the
32         // source object.
33         _data = other._data;
34         _length = other._length;
35
36         // Release the data pointer from the source object so that
37         // the destructor does not free the memory multiple times.
38         other._data = nullptr;
39         other._length = 0;
40     }
41     return *this;
42 }
```

Source: `https://msdn.microsoft.com/en-us/library/dd293665.aspx`.