Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Vectors

- **Static** Data Structures: The array
- From array to vector: `push_back()` and `push_front()`
- Binary Search
- Range Searching
  - Mo's algorithm

The array

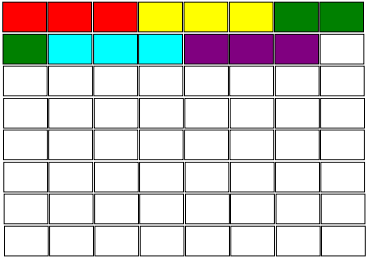
- Allocate enough consecutive space for the data.

$\longrightarrow$ We only need to store the address of the first element.

```
string trivialities[2] = { "this", "class"};
// *trivialities == trivialities[0]
```

Support double indexing.

```
int class = 5;
int exam = 3;
int grades[class][exam];
```



Simple routines

- Swap between two elements. Complexity: $\mathcal{O}(1)$

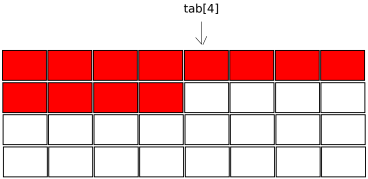
```
void swap(vector<int>& a, int i, int j) {
   int tmp = a[i]; //copy of value a[i]
   a[i] = a[j]; //overriding of a[i]
   a[j] = tmp;
}
```

- Inversion of a sub-vector of length ℓ. Complexity: $\mathcal{O}(\ell)$

```
void invert(vector<int>& a, int first, int last) {
   for(int i = first, j = last; i < j; i++, j--){
      swap(a,i,j)
   }
}
```

The array cont'd

- Pro's
  - Easy Access/Modification of a data
  - "Fast" range searching techniques, even for unsorted data



- Con's
  - **Static** allocation: the (maximal) number of elements must be fixed in advance

Need to resize...

From arrays to **vectors**

For us (and most programming languages...) a vector augments the array data structure with some new operations, in particular:

- **void** `push_back(`**int**`)`. Increases the size by one unit and inserts an element at the end of the array.
- **void** `push_front(`**int**`)`. Increases the size by one unit and inserts an element at the beginning of the array.

Our main algorithmic issue: despite students' dreams, these new operations are not in $\mathcal{O}(1)$. This is because (if there is not enough space available) we need to reallocate all former n elements.

Can we do better?

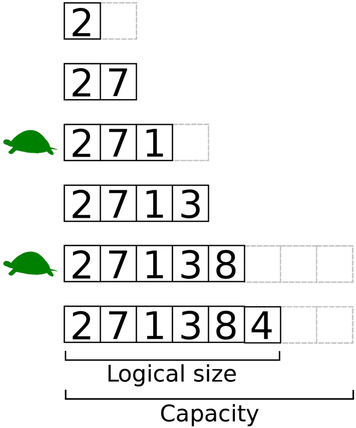
Doubling arrays

Implementation of push_back

- Our n-size array is embedded in some n'-array, for $n' \geq n$.

  $\longrightarrow$ we may store the logical size n in some additional variable.

- push_back in $\mathcal{O}(1)$ as long as $n' > n$.
- If $n' = n$, then we create a bigger $2n$-array and we reallocate.



Amortized complexity: $\mathcal{O}(1)$

Proof: at least $n/2$ elements were inserted since the last time we made a resize.

Potential function: $4\times$ # number of insertions - $\sum$ length of all vectors created

Implementation of `push_back` and `push_front`

- Same idea as before, but our n-size array may not start at 0.

  $\longrightarrow$ Need for another intermediate variable storing the position i_0 of the first element.

- We need to resize if:
  - Either $n' = n$ and a `push_back` occurs;
  - Or $i_0 = 0$ and a `push_front` occurs.

- Whenever we resize, we triple the size of the vector ($n' \longmapsto 3n'$).

  $+$ we embed our n-size vector between pos. n' and $2n'$

$\Longrightarrow$ At least $n'/3$ insertions since the last time we made a resize.

Advanced operations on vectors

Range Searching

Global Input: an n-size vector a[]

Definition

A range query $rq(i, j)$ asks for some information about the elements between positions i and j.

Examples:

- their sum;
- the max./min. element
- searching a value: given some value e, does there exist an integer $i \leq k \leq j$ s.t. $a[k] = e$?
- etc.

Searching a value in a **sorted** vector

Binary Search

Ex: Searching `e = 11` in `[0,1,3,6,11,18,45]`

Searching a value in a **sorted** vector

Binary Search

Ex: Searching `e = 11` in `[0,1,3,6,11,18,45]`

Step 1: Compare `e` with the median

[0,1,3,**6**,11,18,45], we have e > 6

Searching a value in a **sorted** vector

Binary Search

Ex: Searching `e = 11` in `[0,1,3,6,11,18,45]`

Step 1: Compare e with the median

[0,1,3,**6**,11,18,45], we have e > 6

Step 2: Keep searching on the right (*why so?*)

Search e in [11,18,45]

Searching a value in a **sorted** vector

Binary Search

Ex: Searching `e = 11` in `[0,1,3,6,11,18,45]`

Step 1: Compare e with the median

[0,1,3,**6**,11,18,45], we have e > 6

Step 2: Keep searching on the right (*why so?*)

Search e in [11,18,45]

Step 3: [11,**18**,45], e $< 18 \longrightarrow$ Go left

Searching a value in a **sorted** vector

Binary Search

Ex: Searching `e = 11` in `[0,1,3,6,11,18,45]`

Step 1: Compare e with the median

[0,1,3,**6**,11,18,45], we have e > 6

Step 2: Keep searching on the right (*why so?*)

Search e in [11,18,45]

Step 3: [11,**18**,45], e $< 18 \longrightarrow$ Go left

Step 4: [11], e $= 11 \longrightarrow$ `STOP`.

Binary search

Implementation

```
//Searching e between a[l] and a[u]

int dichoSearch(const vector<int>& a, int e, int l, int u) {
    //Case of an empty range
    if(l > u)
        return -1;
    //Computation of the median
    int m = (u+l)/2;
    if(a[m] == e)
        return m;
    else if(a[m] < e)
        return dichoSearch(a,e,m+1,u);
    else
        return dichoSearch(a,e,l,m-1);
}
```

Binary search

Complements

- A powerful method which also applies to “almost sorted” arrays

(more on that during the labs/seminars)

- The index returned by `dichoSearch` may not be the <u>first</u> occurence of the searched element `e` (and not the last one either).

Ex: `[0,1,2,3,`**3**`,3,4,5,13]`, `e = 3` $\longrightarrow$ `m = 4` and `a[m] == e`

Binary search
Complements

- A powerful method which also applies to "almost sorted" arrays

(more on that during the labs/seminars)

- The index returned by `dichoSearch` may not be the first occurence of the searched element `e` (and not the last one either).

Ex: `[0,1,2,3,`**3**`,3,4,5,13]`, `e = 3` $\longrightarrow$ `m = 4` and `a[m] == e`

Solution: Continue the search (left) **but include the median in the range**

`[0,1,2,3,`**3**`,3,4,5,13]` $\rightarrow$ `[0,1,2,3,`**3**`]` $\rightarrow$ `[0,1,`**2**`,3,3]`

$\rightarrow$ `[`**3**`,3]` $\rightarrow$ `[`**3**`]` $\rightarrow$ `STOP`

Variation of Binary Search

Function first()

```
//Searching the first occurence of e between a[l] and a[u]

int first(const vector<int>& a, int e, int l, int u) {
    //Case of an empty range
    if(l > u)
        return -1;
//Case of a single element
    if(l==u)
        return (a[l] == e) ? l : -1;
    //Computation of the median
    int m = (u+l)/2;
    if(a[m] < e)
        return first(a,e,m+1,u);
    else
        return first(a,e,l,m);
}
```

Analysis of Binary search

Question: How many elements in the range considered?

- Step 1: n elements
- Step 2:
- Step 3:
- Step i+1:

Analysis of Binary search

Question: How many elements in the range considered?

- Step 1: n elements
- Step 2: $n/2$ elements
- Step 3:
- Step i+1:

Analysis of Binary search

Question: How many elements in the range considered?

- Step 1: n elements
- Step 2: $n/2$ elements
- Step 3: $n/4$ elements
- Step i+1:

Analysis of Binary search

Question: How many elements in the range considered?

- Step 1: n elements
- Step 2: $n/2$ elements
- Step 3: $n/4$ elements
- Step i+1: $n/2^i$ elements

Analysis of Binary search

Question: How many elements in the range considered?

- Step 1: n elements
- Step 2: $n/2$ elements
- Step 3: $n/4$ elements
- Step i+1: $n/2^i$ elements

Complexity in $\mathcal{O}(\log n)$ (for an array)

Searching a value in an **unsorted** vector

Mo's algorithm

1) Partition the n-vector `a[]` in $\sqrt{n}$ blocks of equal size $n/\sqrt{n} = \sqrt{n}$.

2) Let `b[]` be a copy of `a[]`. Sort each of the $\sqrt{n}$ blocks separately in the copy. Pre-processing time: $\mathcal{O}(n) + \sqrt{n} \times \mathcal{O}(\sqrt{n}^2) = \mathcal{O}(n\sqrt{n})$.

3) Searching value e between pos. i and j.

- Let $B_1, B_2, \ldots, B_r$ be the blocks fully between i and j. Binary search in their sorted copies.
- The block containing $a[i]$ (resp., $a[j]$) may start before i (resp., end after j). Linear search of value e amongst the $\leq \sqrt{n}$ elements in this block that are between pos. i and j.

Query time: $\mathcal{O}(\sqrt{n}) + \mathcal{O}(r \cdot \log n) = \mathcal{O}(\sqrt{n} \log n)$.

Example

Input: `a[] = {3,44,1,7,23,19,0,101,89}`

Sorted copy: `b[] = {1,3,44,7,19,23,0,89,101}`

- Search for some value e between $i = 1$ and $j = 6$.
  - Binary search in the block $7, 19, 23$ (fully between i and j)
  - Exhaustive search in the partial block $44, 1$
  - Exhaustive search in the partial block 0.

Remark: Exhaustive Search in `a[]`. Binary search in `b[]`.

Another example: Sum of elements

Mo's algorithm + dynamic programming:

1) In an auxiliary $\sqrt{n}$-size vector `c[]`, store the sum of all elements within the same block.

Ex.: if `a[] = {3,44,1,7,23,19,0,101,89}` then `c[] = {48,49,190}`.

3) Sum of all elements between pos. i and j.

- Let $B_1, B_2, \ldots, B_r$ be the blocks fully between i and j. Sum the pre-computed values for these blocks (in `c[]`).
- The block containing $a[i]$ (resp., $a[j]$) may start before i (resp., end after j). Sum of the $\leq \sqrt{n}$ elements in this block that are between pos. i and j.

Ex. $i = 1$, $j = 6$. The sum equals 44+1+c[1]+0.

Sum of elements: Comparison between two methods

- Using Mo's algorithm + dynamic programming.
  - Pre-processing in $\mathcal{O}(n)$ time and space;
  - Query time in $\mathcal{O}(\sqrt{n})$;
  - If an element of the vector `a[]` is modified, then we can update the vector `c[]` in $\mathcal{O}(1)$ (at most one block is impacted)
- **Alternative method**: Store in an auxiliary vector the values `b[i] =` $\sum_{k=0}^{i} a[k]$. Pre-processing: $\mathcal{O}(n)$
  - Query time in $\mathcal{O}(1)$! – Return `b[j] - b[i]+a[i]`
  - But if an element of the vector `a[]` is modified, then updating the vector `b[]` may require $\mathcal{O}(n)$ time

Questions