Chapter 2. Divide-and-conquer algorithms

Data Structures and Algorithms

FIB

Q2 2018–19

Jordi Delgado
(slides by Antoni Lozano)

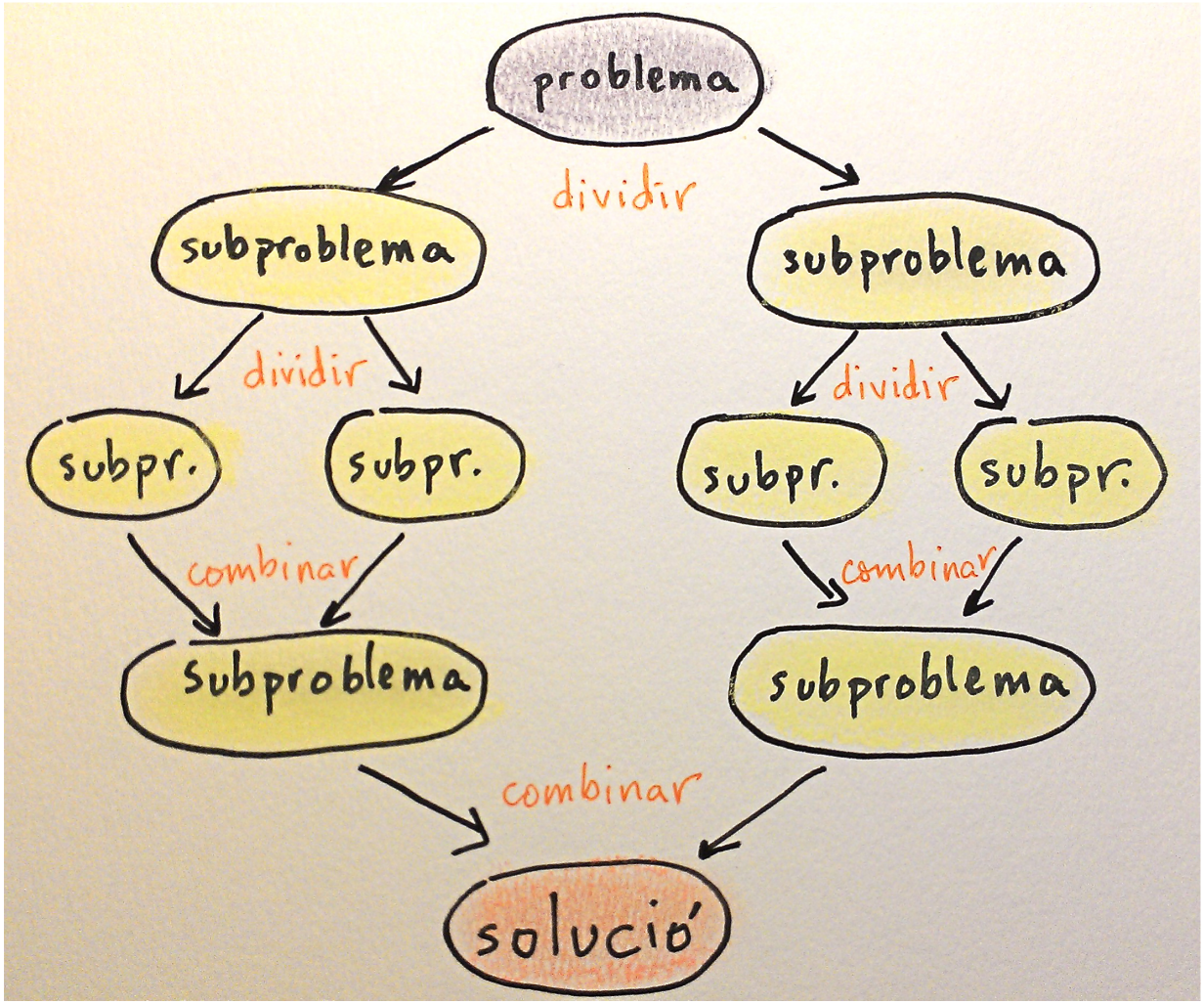
Chapter 2. Divide-and-conquer algorithms

Divide and conquer

Divide and conquer solves a problem in three steps:

1. dividing it into *subproblems* (smaller instances of the same type of problem)
2. recursively solving the subproblems and
3. combining the answers in the right way

Divide and conquer



Hence, work is done in three parts: (1) at the division into subproblems, (2) at the base case of the recursion (3) at the recombination of solutions

Divide and conquer

Divide-and-conquer algorithms usually follow a uniform strategy.
A problem of size n is solved:

- dividing it into a subproblems of size n/b and
- combining the answers in time n^k,

where a, b, k are natural numbers. In this situation, the cost can be expressed with the recurrence:

$$T(n) = a \cdot T(n/b) + \Theta(n^k)$$

that can be solved using master theorem II.

Divide and conquer

```
int binary_search(const vector<int>& a,int i,int j,int v)
{  if (i <= j) {
       int k = (i + j) / 2;
       if (v == a[k])
           return k;
       else if (v < a[k])
           return binary_search(a, i, k-1, v);
       else
           return binary_search(a, k+1, j, v);
   } return -1;
}
```

The parameter of recursion is $n = j - i$ and the cost $T(n) = T(n/2) + \Theta(1)$. Using master theorem II, $T(n) \in \Theta(\log n)$.

Divide and conquer

Exercise 2.2

Range. Write a divide-and-conquer based algorithm of cost $\Theta(\log n)$ such that, given a sorted table T with n different elements and two elements x and y with $x \leq y$, returns the number of elements in T that are between x and y (x and y are included).

Chapter 2. Divide-and-conquer algorithms

Basic mergesort

Mergesort is a good example of a divide-and-conquer algorithm. It uses an almost optimal number of comparisons and it is stable in the following sense:

- it preserves the order between equal elements
- its behavior is similar independently of the sortedness level of the input

Mergesort was invented by John von Neumann in 1945.

Basic mergesort

The main operation combines (or merges) two sorted vectors into one. Given a vector T of size ≥ 2, the algorithm:

1. Recursively sorts the first half of T.
2. Recursively sorts the second half of T.
3. Merges the two halves.

Basic mergesort

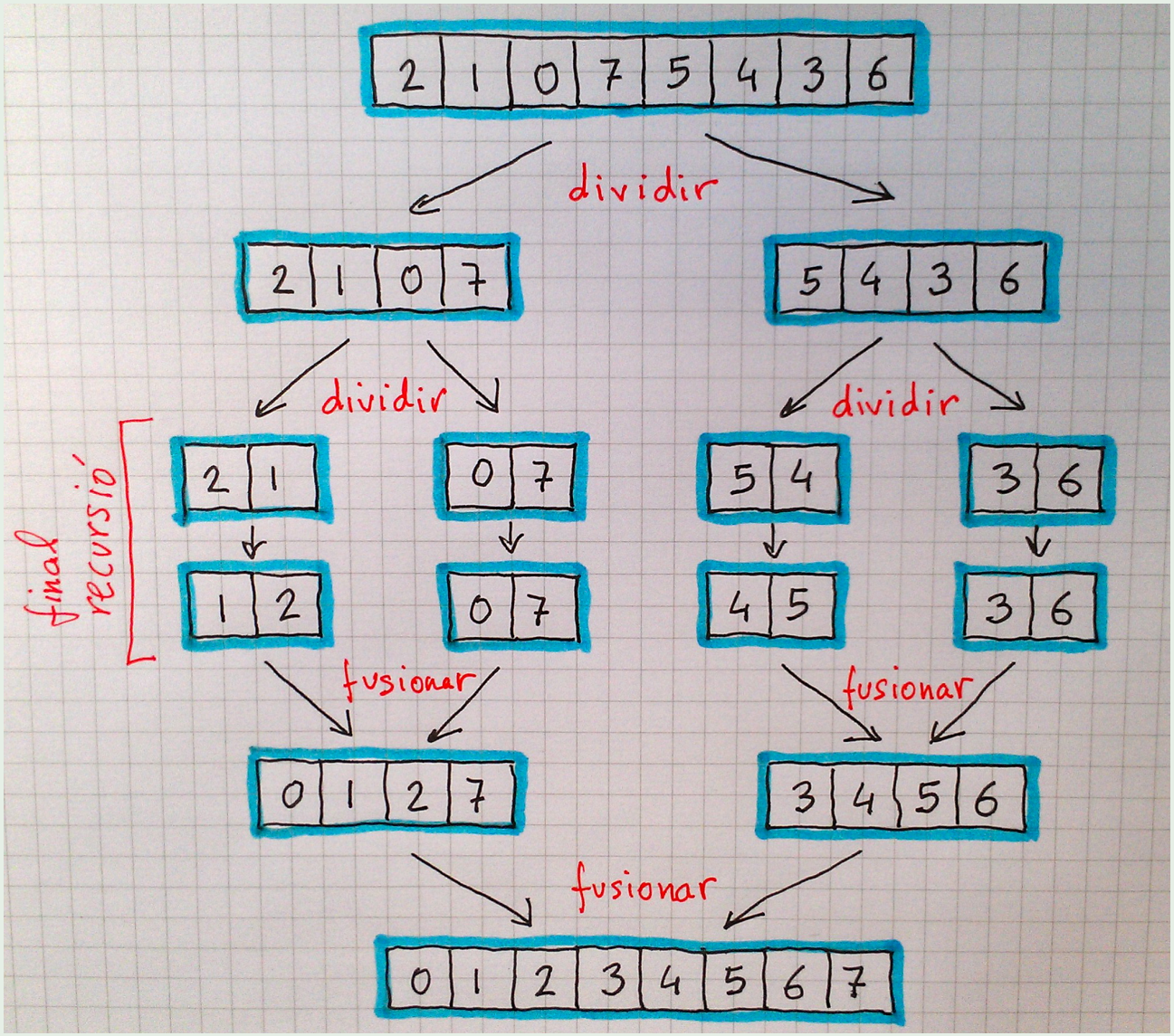
Mergesort (*Algorismes en C++, EDA*)

```
template <typename elem>
void mergesort (vector<elem>& T) {
    mergesort(T, 0, T.size() - 1);
}

template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    if (e<d) {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
}   }
```

Basic mergesort

Example (here, recursion ends with size 2, in the algorithm with size 1)



Basic mergesort

One of the key parts of the algorithm consists in merging two sorted vectors.

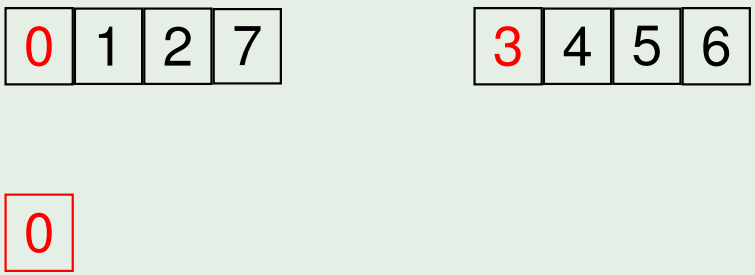
Merge (*Algorismes en C++, EDA*)

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];
    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];
}
```

Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

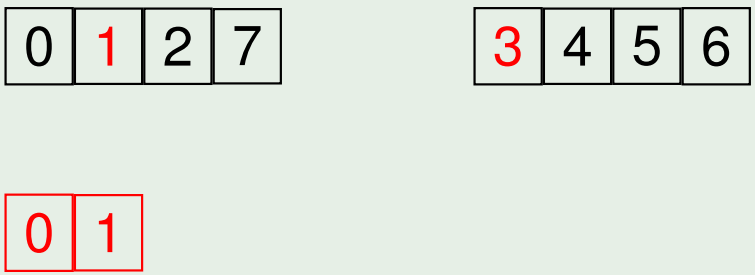
Example



Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

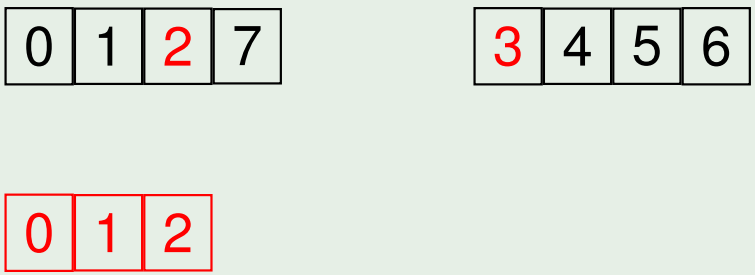
Example



Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

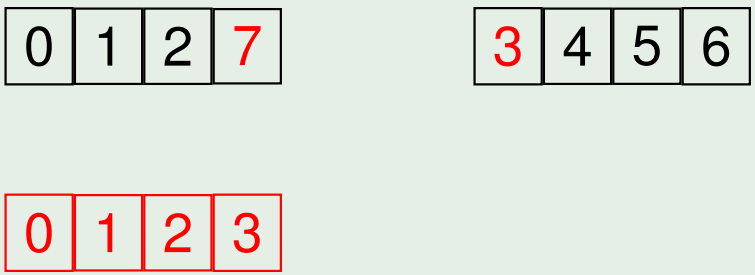
Example



Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

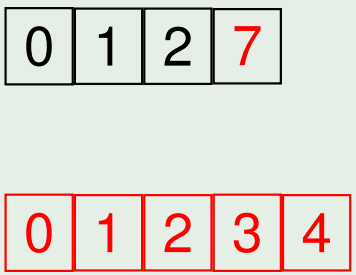
Example



Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

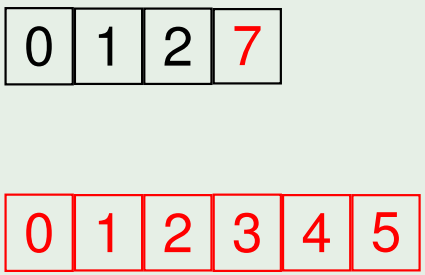
Example



Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

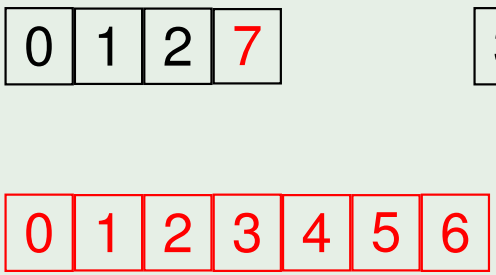
Example



Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

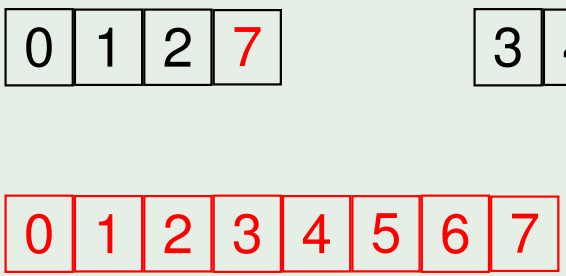
Example



Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];    }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

Example



Basic mergesort

```
template <typename elem>
void merge (vector<elem>& T, int e, int m, int d) {
    vector<elem> B(d - e + 1);
    int i = e, j = m + 1, k = 0;
    while (i <= m and j <= d) {
        if (T[i] <= T[j]) B[k++] = T[i++];
        else B[k++] = T[j++];   }
    while (i <= m) B[k++] = T[i++];
    while (j <= d) B[k++] = T[j++];
    for (k = 0; k <= d-e; ++k) T[e + k] = B[k];   }
```

Remark

Each comparison adds an element to B except for the last one, that adds at least two.

- Hence, the number of comparisons is $< N = d - e$.
- The number of assignments is $2N$.
- The cost is linear.

Basic mergesort

```
template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    if (e<d) {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
}   }
```

Since `merge` is linear, the cost of mergesort can be easily expressed with the recurrence:

$$T(1) = \Theta(1)$$

$$T(n) = 2T(n/2) + \Theta(n) \quad \text{for } n > 1$$

and, using master theorem II, we have that

$$T(n) \in \Theta(n \log n).$$

Variants

Mergesort with insertion sort for small vectors (*Alg. en C++, EDA*)

```
template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    const int critical_size = 50;
    if (d-e < critical_size)
        insertion_sort(T, e, d);
    else {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
}   }
```

Variants

The two iterative versions that we will see are based on the fact that merges are done at the end of the recursion. Hence:

- they start directly with the elements to sort and
- they sort the vector by means of merges

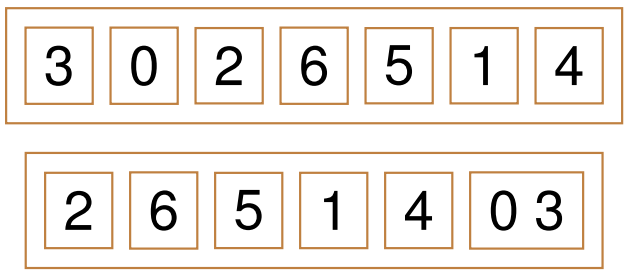
Variants

Iterative mergesort 1

Version from the book *Algorithms* by Dasgupta/Papadimitriou/Vazirani in pseudocode (pag. 51). It uses an abstract data type queue with operations:

- `inject(Q,e)`: adds element `e` to the queue `Q`
- `eject(Q)`: removes and returns the last element in `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (empty queue)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```



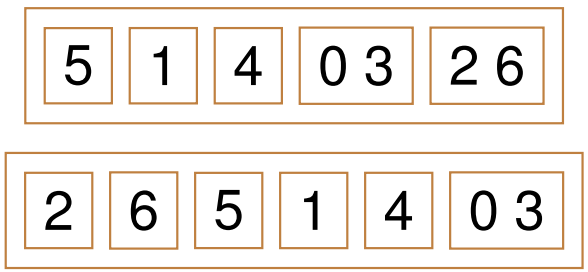
Variants

Iterative mergesort 1

Version from the book *Algorithms* by Dasgupta/Papadimitriou/Vazirani in pseudocode (pag. 51). It uses an abstract data type queue with operations:

- `inject(Q,e)`: adds element `e` to the queue `Q`
- `eject(Q)`: removes and returns the last element in `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (empty queue)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```

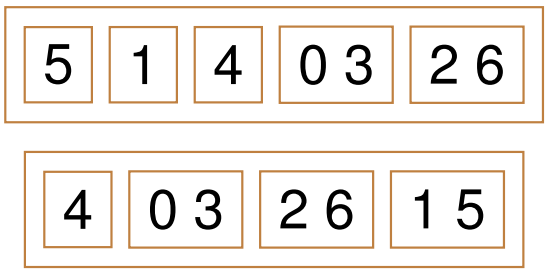


Iterative mergesort 1

Version from the book *Algorithms* by Dasgupta/Papadimitriou/Vazirani in pseudocode (pag. 51). It uses an abstract data type queue with operations:

- `inject(Q,e)`: adds element `e` to the queue `Q`
- `eject(Q)`: removes and returns the last element in `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (empty queue)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```

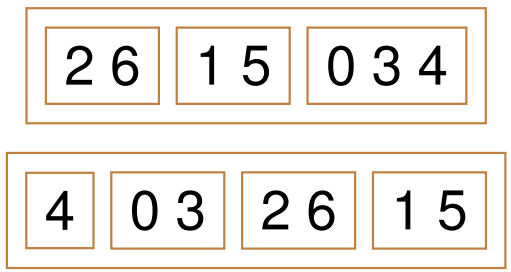


Iterative mergesort 1

Version from the book *Algorithms* by Dasgupta/Papadimitriou/Vazirani in pseudocode (pag. 51). It uses an abstract data type queue with operations:

- `inject(Q,e)`: adds element `e` to the queue `Q`
- `eject(Q)`: removes and returns the last element in `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (empty queue)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```



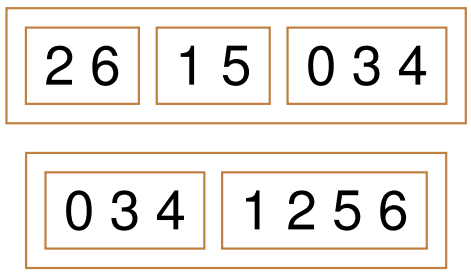
Variants

Iterative mergesort 1

Version from the book *Algorithms* by Dasgupta/Papadimitriou/Vazirani in pseudocode (pag. 51). It uses an abstract data type queue with operations:

- `inject(Q,e)`: adds element `e` to the queue `Q`
- `eject(Q)`: removes and returns the last element in `Q`

```
function mergesort_queue(a[1...n])
    Q = [] (empty queue)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```



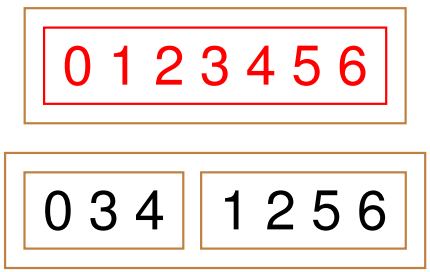
Variants

Iterative mergesort 1

Version from the book *Algorithms* by Dasgupta/Papadimitriou/Vazirani in pseudocode (pag. 51). It uses an abstract data type queue with operations:

- `inject(Q,e)`: adds element `e` to the queue `Q`
- `eject(Q)`: removes and returns the last element in `Q`

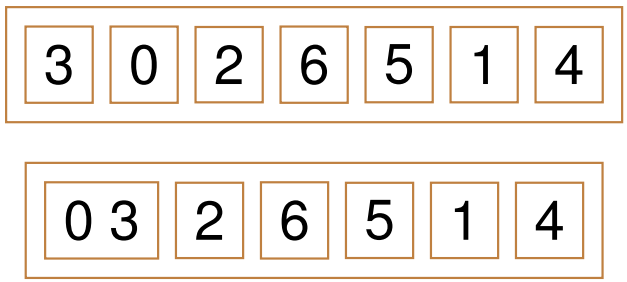
```
function mergesort_queue(a[1...n])
    Q = [] (empty queue)
    for i=1 to n:
        inject(Q, a[i])
    while |Q| > 1:
        inject(Q, merge(eject(Q), eject(Q)))
    return eject(Q)
```



Variants

Iterative mergesort 2 (Algorismes en C++, EDA)

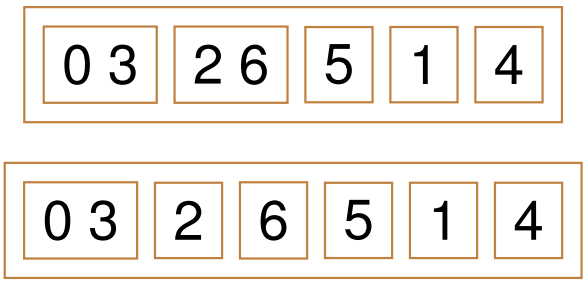
```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
 int n = T.size();
    for (int m = 1; m < n; m *= 2) {
       for (int i = 0; i < n-m; i += 2*m) {
          merge(T, i, i+m-1, min(i+2*m-1, n-1));
}   }   }
```



Variants

Iterative mergesort 2 (Algorismes en C++, EDA)

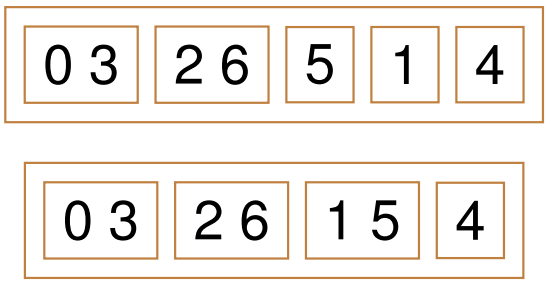
```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
 int n = T.size();
    for (int m = 1; m < n; m *= 2) {
       for (int i = 0; i < n-m; i += 2*m) {
          merge(T, i, i+m-1, min(i+2*m-1, n-1));
}   }   }
```



Variants

Iterative mergesort 2 (Algorismes en C++, EDA)

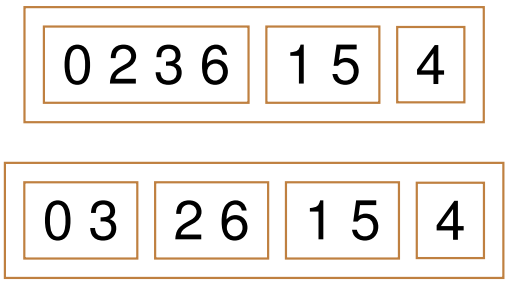
```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
 int n = T.size();
    for (int m = 1; m < n; m *= 2) {
       for (int i = 0; i < n-m; i += 2*m) {
          merge(T, i, i+m-1, min(i+2*m-1, n-1));
}   }   }
```



Variants

Iterative mergesort 2 (Algorismes en C++, EDA)

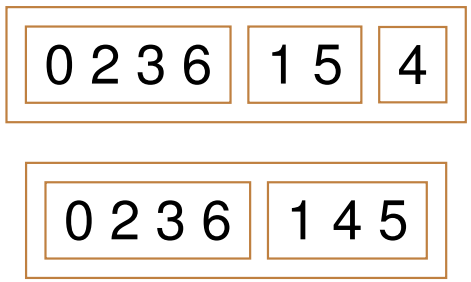
```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
 int n = T.size();
    for (int m = 1; m < n; m *= 2) {
       for (int i = 0; i < n-m; i += 2*m) {
          merge(T, i, i+m-1, min(i+2*m-1, n-1));
}   }   }
```



Variants

Iterative mergesort 2 (Algorismes en C++, EDA)

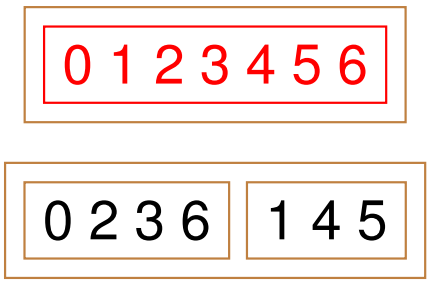
```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
 int n = T.size();
    for (int m = 1; m < n; m *= 2) {
       for (int i = 0; i < n-m; i += 2*m) {
          merge(T, i, i+m-1, min(i+2*m-1, n-1));
}   }   }
```



Variants

Iterative mergesort 2 (Algorismes en C++, EDA)

```
template <typename elem>
void mergesort_bottom_up (vector<elem>& T) {
 int n = T.size();
    for (int m = 1; m < n; m *= 2) {
       for (int i = 0; i < n-m; i += 2*m) {
          merge(T, i, i+m-1, min(i+2*m-1, n-1));
}   }   }
```



Exercises

Exercise 2.3

Bottom-up, top-down. Sort the table $\langle 3, 8, 15, 7, 12, 6, 5, 4, 3, 7, 1 \rangle$ with recursive mergesort and with the bottom-up iterative version (2).

Exercises

Remember recursive *mergesort*

```
template <typename elem>
void mergesort (vector<elem>& T) {
    mergesort(T, 0, T.size() - 1);
}

template <typename elem>
void mergesort (vector<elem>& T, int e, int d) {
    if (e<d) {
        int m = (e + d) / 2;
        mergesort(T, e, m);
        mergesort(T, m + 1, d);
        merge(T, e, m, d);
}   }
```

Exercise 2.5

Stack height. Which is the maximum number of recursive calls to be stored in the stack in order to sort a table with n elements?

Chapter 2. Divide-and-conquer algorithms

General algorithm

As it name indicates, *quicksort* is the quickest generic sorting algorithm. Even though its worst-case cost is $\Theta(n^2)$, average case is $\Theta(n \log n)$, and the efficiency of its internal loop makes it the algorithm that better behaves in practice.

Quicksort was invented C. A. R. Hoare in 1960.

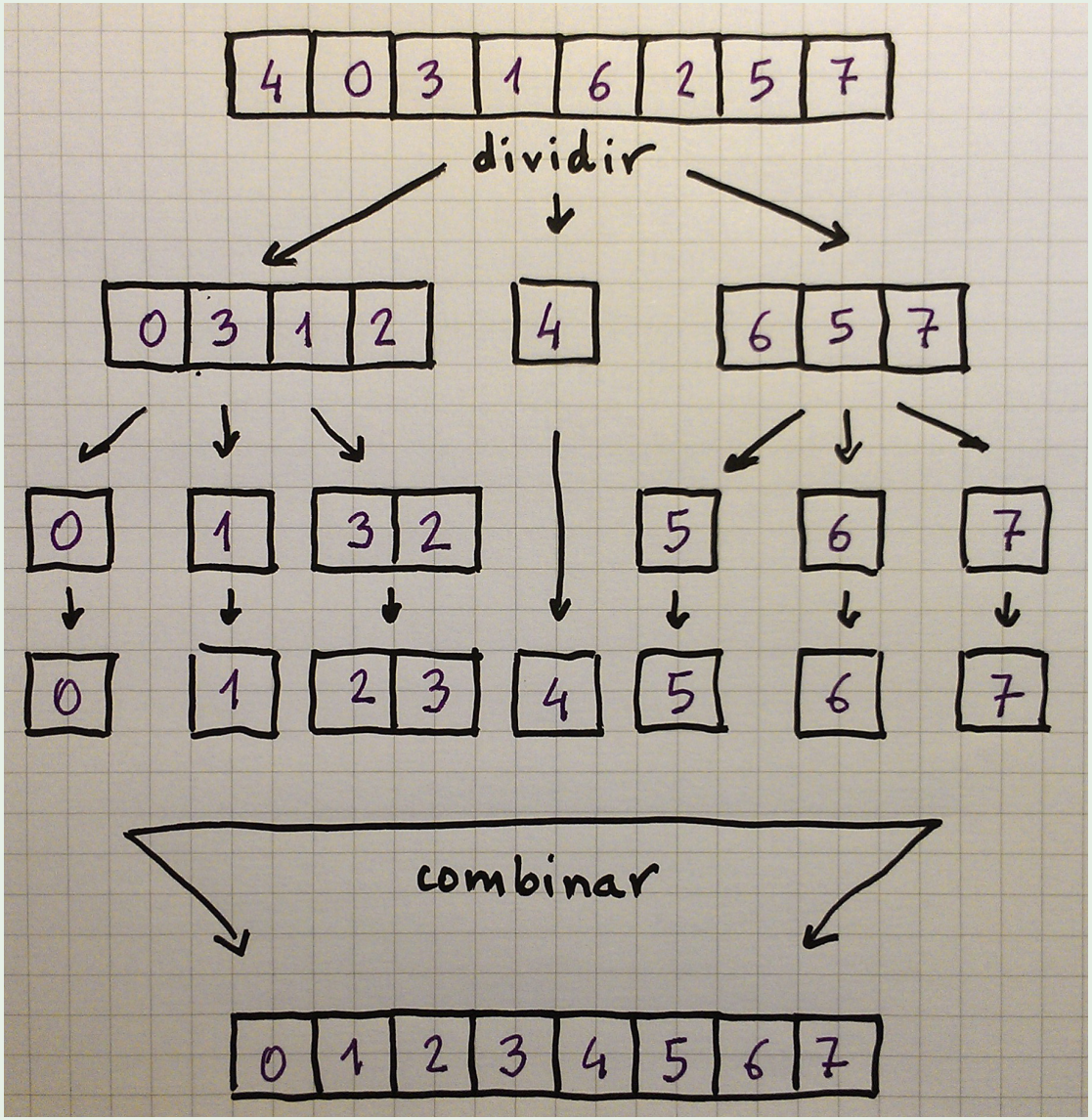
General algorithm

Given a vector T with at least 2 elements, the 'basic algorithm does the following:

1. Choose an element x in T.
2. Divides $T - \{x\}$ in two disjoint groups:
   - T_1, containing all elements $\leq x$ in T
   - T_2, containing all elements $\geq x$ in T.
3. Sort T_1 and T_2 recursively.
4. Return T_1 followed by x and T_2.

General algorithm

Example



General algorithm

Quicksort (*Algorismes en C++, EDA*)

```
void quicksort(vector<elem>& T) {
    quicksort(T, 0, T.size() - 1);
}

template <typename elem>
void quicksort(vector<elem>& T, int e, int d) {
    if (e < d) {
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
}   }
```

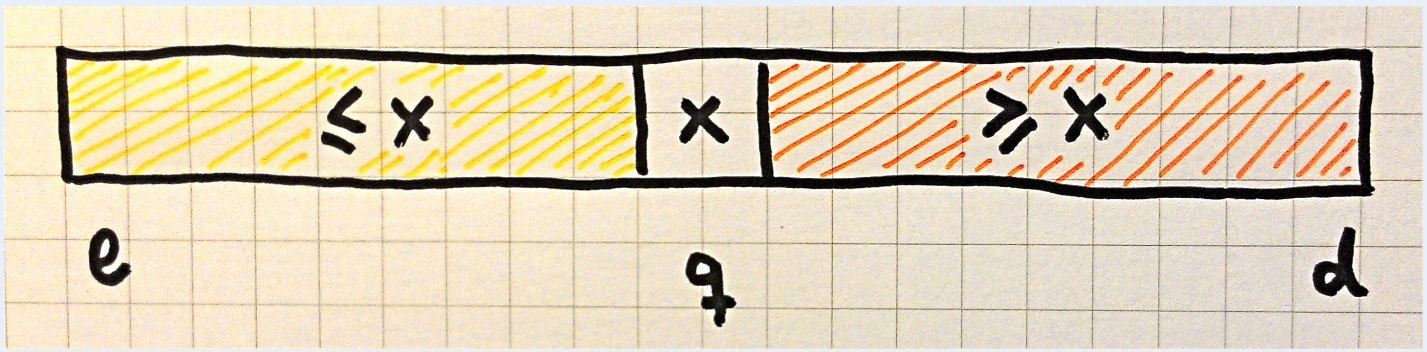
- Precondition of `q = partition(T, e, d)`: $0 \leq e \leq d \leq T.size() - 1$
- Postcondition of `q = partition(T, e, d)`: for all i
  - if $e \leq i \leq q$, we have that $T[i] \leq T[q]$
  - if $q \leq i \leq d$, we have that $T[i] \geq T[q]$

General algorithm

Quicksort (*Algorismes en C++, EDA*)

```
void quicksort(vector<elem>& T) {
    quicksort(T, 0, T.size() - 1);
}

template <typename elem>
void quicksort(vector<elem>& T, int e, int d) {
    if (e < d) {
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
}   }
```



Comparison between mergesort and quicksort

- Similarities with mergesort:
  - solves two subproblems
  - performs linear additional work
- Differences:
  - Mergesort: division into subproblems is trivial, merge is not
  - Quicksort: division into subproblems is non-trivial, combination is

Hoare's partition

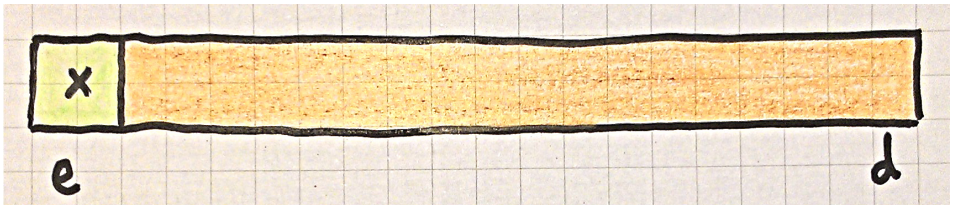
Original Hoare's partition with the first element as a pivot

Hoare's partition (*Algorismes en C++, EDA*)

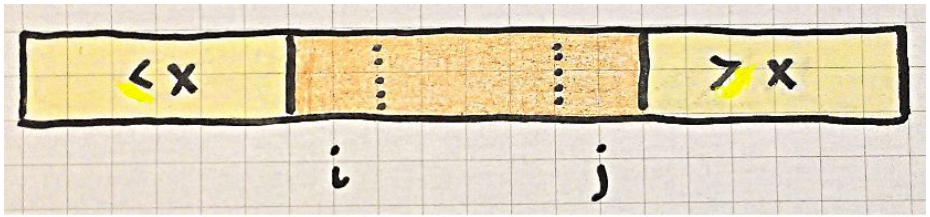
```
template <typename elem>
int partition (vector<elem>& T, int e, int d) {
    elem x = T[e];
    int i = e - 1;
    int j = d + 1;
    for (;;) {
        while (x < T[--j]);
        while (T[++i] < x);
        if (i >= j) return j;
        swap(T[i], T[j]);
}   }
```

Hoare's partition

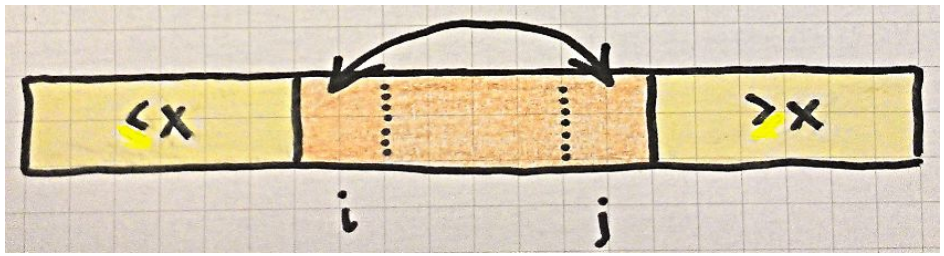
- Start of the function `partition(T, e, d)`:



- Main loop:
  - look for the most central values i, j such that



  - swap the contents of positions i, j



Variants

- Another difference with mergesort is that subproblems might have different size.
- Subproblems size depends on the choice of the initial element, called pivot.

Variants

We consider three strategies for choosing the pivot:

1. Choose the first element:
   - acceptable if input is random
   - if input is sorted (increasingly or decreasingly), algorithm takes $\Theta(n^2)$ time to do nothing
2. Choose a random element
   - safe choice to divide into similar subproblems
   - algorithm might not be faster due to the cost of generating random numbers
3. Choose the median of three elements
   - the best option would be to choose the median of the vector, but it is too expensive
   - a good estimation is to use the median of 3, usually the 1st, the one in the middle and the last one

Variants

We consider three strategies for choosing the pivot:

1. Choose the first element:
   - acceptable if input is random
   - if input is sorted (increasingly or decreasingly), algorithm takes $\Theta(n^2)$ time to do nothing
2. Choose a random element
   - safe choice to divide into similar subproblems
   - algorithm might not be faster due to the cost of generating random numbers
3. Choose the median of three elements
   - the best option would be to choose the median of the vector, but it is too expensive
   - a good estimation is to use the median of 3, usually the 1st, the one in the middle and the last one

Variants

We consider three strategies for choosing the pivot:

1. Choose the first element:
   - acceptable if input is random
   - if input is sorted (increasingly or decreasingly), algorithm takes $\Theta(n^2)$ time to do nothing
2. Choose a random element
   - safe choice to divide into similar subproblems
   - algorithm might not be faster due to the cost of generating random numbers
3. Choose the median of three elements
   - the best option would be to choose the median of the vector, but it is too expensive
   - a good estimation is to use the median of 3, usually the 1st, the one in the middle and the last one

Variants

Quicksort with random pivot (*Algorismes en C++, EDA*)

```
template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    if (e < d) {
        int p = randint(e, d);
        swap(T[e], T[p]);
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
} }
```

Variants

Quicksort with the median as the pivot

```
   template <typename elem>
   void quicksort (vector<elem>& T, int e, int d) {
       if (e < d) {
           int center = ( e + d ) / 2;
1          if (T[e] < T[center]) swap(T[center], T[e]);
2          if (T[d] < T[center]) swap(T[center], T[d]);
3          if (T[d] < T[e])      swap(T[e], T[d]);
           // el pivot is in position e
           int q = partition(T, e, d);
           quicksort(T, e, q);
           quicksort(T, q + 1, d);
   }   }
```

After lines 1, 2 and 3:

$$\text{T[center]} \leq \text{T[e]}, \quad \text{T[center]} \leq \text{T[d]}, \quad \text{T[e]} \leq \text{T[d]}.$$

Hence, T[center] $\leq$ T[e] $\leq$ T[d] and the median is T[e].

Variants

For small vectors, insertion sort behaves better than *quicksort*. Hence, a good solution is to cut the recursion when the vector size is small enough (usually, size between 5 and 20).

Quicksort with insertion sort for small vectors

```
template <typename elem>
void quicksort (vector<elem>& T, int e, int d) {
    const int critical_size = 20;
    if (d - e < critical_size)
       ordena_insercio(T, e, d);
    else {
        int q = partition(T, e, d);
        quicksort(T, e, q);
        quicksort(T, q + 1, d);
}  }
```

Analysis

Recurrence

Let $T(n)$ be the cost of quicksort on n elements. Then,

- $T(0), T(1) \in \Theta(1)$
- For $n \geq 2$,

$$T(n) = T(i) + T(n - i - 1) + \Theta(n),$$

where i is the number of elements smaller than the pivot.

Analysis: worst case

General case

$$T(n) = T(i) + T(n-i-1) + \Theta(n)$$

In the worst case, the pivot is always the smallest or the largest element. As a result, one recursive call has constant cost and the other one $T(n-1)$.

Worst case

$$T(n) = T(n-1) + \Theta(n)$$

Solving the recurrence we obtain:

$$T(n) \in \Theta(n^2).$$

Analysis: worst case

General case

$$T(n) = T(i) + T(n - i - 1) + \Theta(n)$$

In the worst case, the pivot is always the smallest or the largest element. As a result, one recursive call has constant cost and the other one $T(n-1)$.

Worst case

$$T(n) = T(n-1) + \Theta(n)$$

Solving the recurrence we obtain:

$$T(n) \in \Theta(n^2).$$

Analysis: best case

General case

$$T(n) = T(i) + T(n-i-1) + \Theta(n)$$

In the best case, the pivot is the median of the vector and hence, both subvectors have the same size (the difference of ≤ 1 is not relevant for Θ).

Best case

$$T(n) = 2T(n/2) + \Theta(n)$$

It is the same recurrence as for mergesort. Hence, its cost is:

$$T(n) \in \Theta(n \log n).$$

Analysis: best case

General case

$$T(n) = T(i) + T(n-i-1) + \Theta(n)$$

In the best case, the pivot is the median of the vector and hence, both subvectors have the same size (the difference of ≤ 1 is not relevant for Θ).

Best case

$$T(n) = 2T(n/2) + \Theta(n)$$

It is the same recurrence as for mergesort. Hence, its cost is:

$$T(n) \in \Theta(n \log n).$$

Analysis: average case

General case

$$T(n) = T(i) + T(n-i-1) + \Theta(n)$$

For the average case, we assume that the partition is random and hence, each one has probability $1/n$. The average value of $T(i)$ and $T(n-i-1)$ will be of $\frac{1}{n}\sum_{j=0}^{n-1} T(j)$.

Average case

For some constant c,

$$T(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} T(j)\right] + cn \tag{1}$$

Analysis: average case

General case

$$T(n) = T(i) + T(n-i-1) + \Theta(n)$$

For the average case, we assume that the partition is random and hence, each one has probability $1/n$. The average value of $T(i)$ and $T(n-i-1)$ will be of $\frac{1}{n}\sum_{j=0}^{n-1} T(j)$.

Average case

For some constant c,

$$T(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} T(j)\right] + cn \tag{1}$$

Analysis: average case

We multiply equation 1 by n:

$$nT(n) = 2\left[\sum_{j=0}^{n-1} T(j)\right] + cn^2. \tag{2}$$

To remove the sum, we write case $n-1$

$$(n-1)T(n-1) = 2\left[\sum_{j=0}^{n-2} T(j)\right] + c(n-1)^2 \tag{3}$$

and subtract equation 3 from equation 2

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c. \tag{4}$$

Reordering terms and removing c,

$$nT(n) = (n+1)T(n-1) + 2cn. \tag{5}$$

Analysis: average case

We divide equation 5 by $n(n+1)$:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}. \tag{6}$$

We replace n for all values from $n-1$ to 2:

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}. \tag{7}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2c}{n-1}. \tag{8}$$

$$\vdots$$

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}. \tag{9}$$

The sum of all equations 6–9 is

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c\sum_{i=3}^{n+1}\frac{1}{i}. \tag{10}$$

Analysis: average case

On the other hand, we know that

$$\sum_{i=3}^{n+1} \frac{1}{i} = \ln(n+1) + \gamma - \frac{3}{2},$$

where $\gamma \approx 0.577$ is Euler's constant. Replacing this value in equation 10

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \sum_{i=3}^{n+1} \frac{1}{i}$$

we deduce that

$$\frac{T(n)}{n+1} \in \Theta(\log n)$$

and hence

$$T(n) \in \Theta(n \log n).$$

Exercises

Exercise 2.6

***Quicksort* by hand.** Sort the table $\langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$ with quicksort and Hoare's partition.

Exercise 2.7

***Quicksort*.** Which is the cost of quicksort with Hoare's partition when the input:

- is random
- is sorted in increasing order
- is sorted in decreasing order
- has all elements equal

Exercises

Exercise 2.6

***Quicksort* by hand.** Sort the table $\langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$ with quicksort and Hoare's partition.

Exercise 2.7

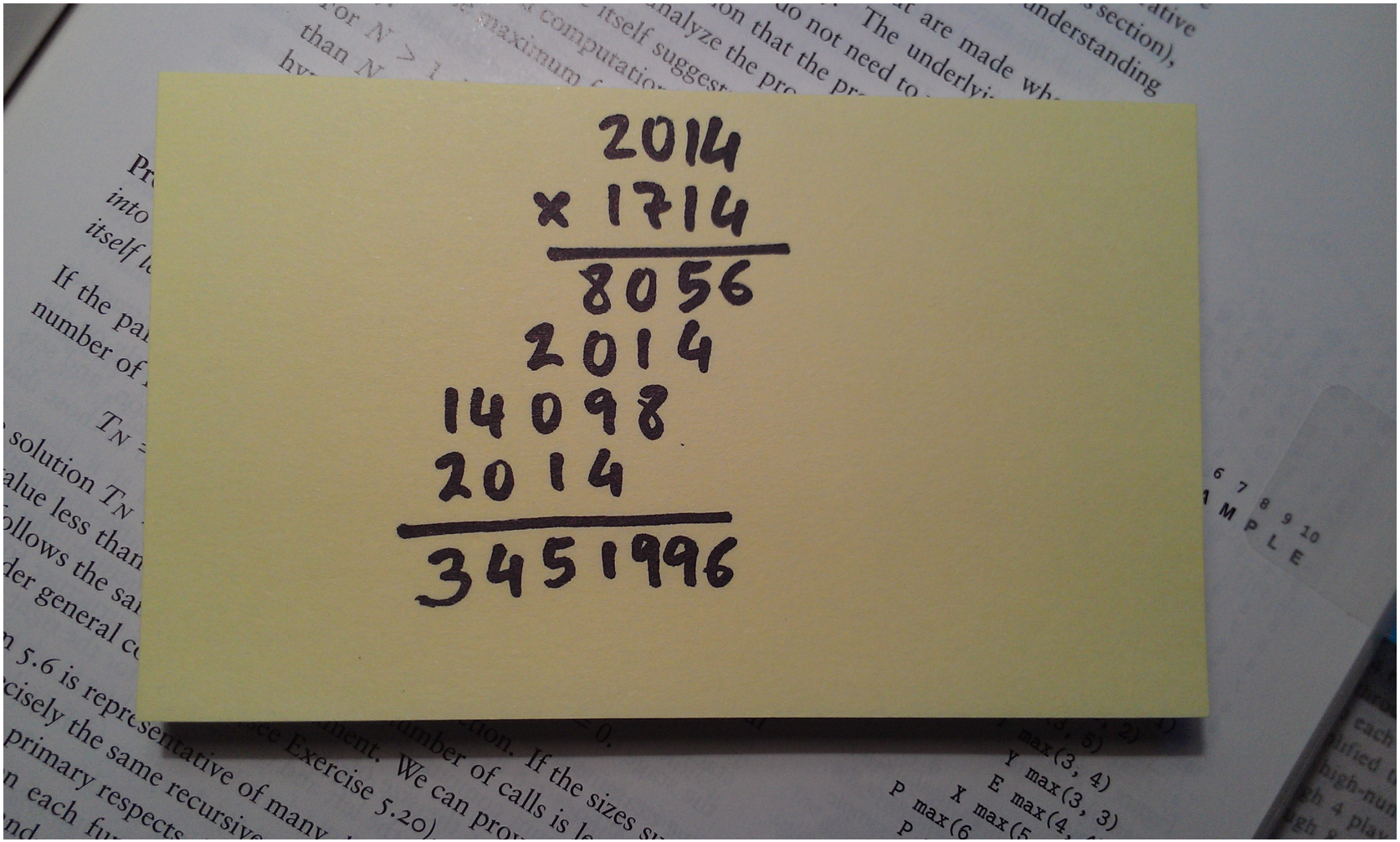
***Quicksort*.** Which is the cost of quicksort with Hoare's partition when the input:

- is random
- is sorted in increasing order
- is sorted in decreasing order
- has all elements equal

Chapter 2. Divide-and-conquer algorithms

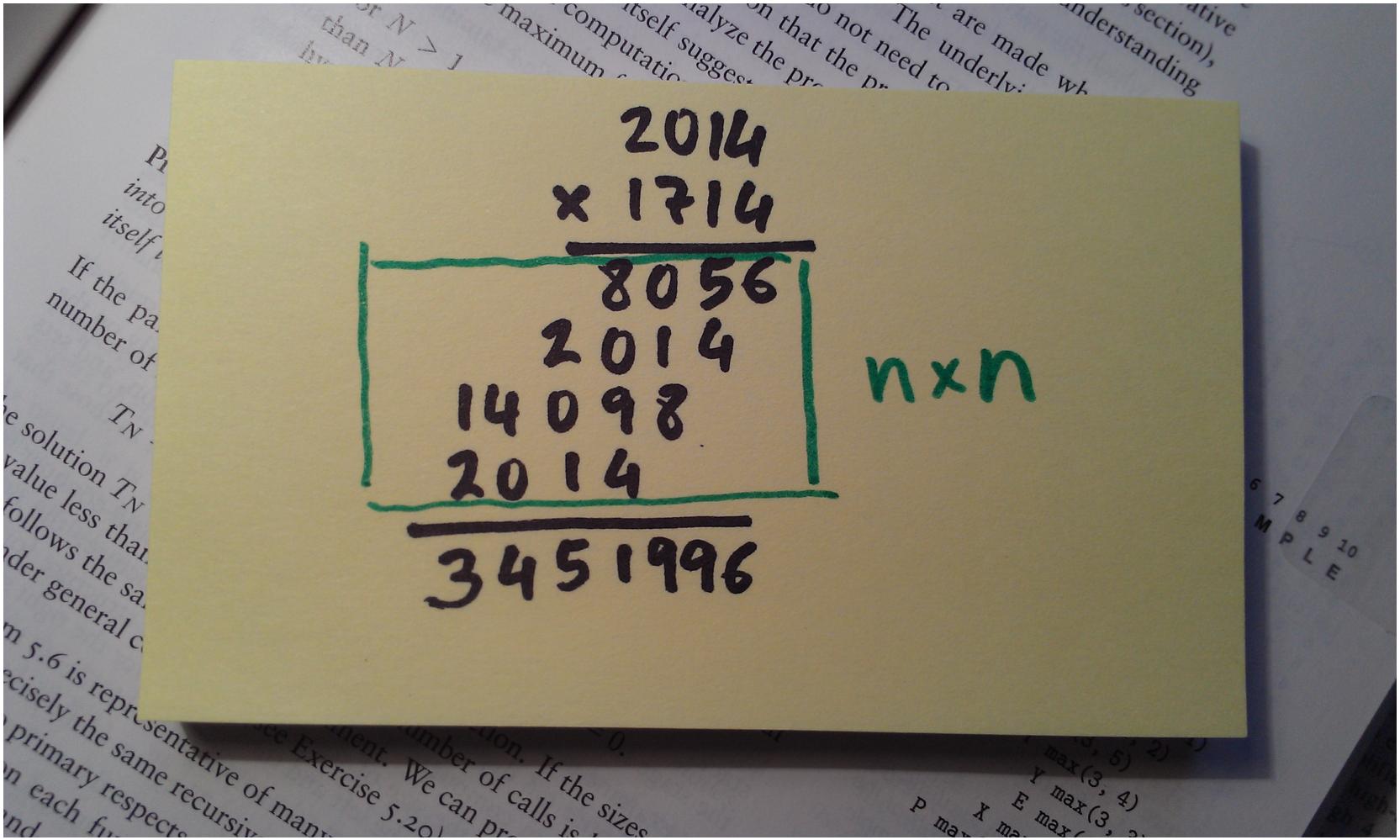
Karatsuba's algorithm

Which is the cost of the standard multiplication algorithm?



Karatsuba's algorithm

$\Theta(n^2)$, where n is the number of digits of the largest number.



Karatsuba's algorithm

Russian mathematician A. Kolmogorov conjectured that such standard algorithm is optimal.

Conjecture (Kolmogorov, 1952)

Any algorithm for multiplying two numbers of n digits has cost $\Omega(n^2)$.

A 23-year Kolmogorov's student, Anatolii Alexeevitch Karatsuba, found an algorithm with cost $\Theta(n^{1.585})$.

Refutation (Karatsuba, 1960)

There is an algorithm that multiplies two numbers of n digits in time $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.

Karatsuba's algorithm

Russian mathematician A. Kolmogorov conjectured that such standard algorithm is optimal.

Conjecture (Kolmogorov, 1952)

Any algorithm for multiplying two numbers of n digits has cost $\Omega(n^2)$.

A 23-year Kolmogorov's student, Anatolii Alexeevitch Karatsuba, found an algorithm with cost $\Theta(n^{1.585})$.

Refutation (Karatsuba, 1960)

There is an algorithm that multiplies two numbers of n digits in time $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$.

Karatsuba's algorithm

The origin of the idea of Karatsuba's algorithm dates back to the XVIII century.

Remark

Mathematician Carl Friedrich Gauss (1777-1855) found out that even though the product of two complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

seems to need 4 products of real numbers, it can be obtained with only 3. The reason is that

$$bc + ad = (a + b)(c + d) - ac - bd.$$

Karatsuba's algorithm

Let us assume that x and y are two integers of n bits. We can express x, y in two parts:

$$x = \boxed{x_E}\,\boxed{x_D} = 2^{\lfloor n/2 \rfloor} x_E + x_D$$

$$y = \boxed{y_E}\,\boxed{y_D} = 2^{\lfloor n/2 \rfloor} y_E + y_D$$

Example

If $x = 10010111_2$ and $y = 11001010_2$ (subindex 2 means ïn binary"), then

$$x = \boxed{x_E}\,\boxed{x_D} = \boxed{1001_2}\,\boxed{0111_2}$$

$$y = \boxed{y_E}\,\boxed{y_D} = \boxed{1100_2}\,\boxed{1010_2}$$

Karatsuba's algorithm

Now, the product

$$xy = (2^{\lfloor n/2 \rfloor} x_E + x_D)(2^{\lfloor n/2 \rfloor} y_E + y_D)$$

when n is even can be rewritten as

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D \tag{11}$$

and when n is odd as

$$xy = 2^{n-1} x_E y_E + 2^{\lfloor n/2 \rfloor}(x_E y_D + x_D y_E) + x_D y_D.$$

An algorithm based on this expression has a cost of

$$T(n) = 4T(n/2) + \Theta(n).$$

Karatsuba's algorithm

Now, the product

$$xy = (2^{\lfloor n/2 \rfloor} x_E + x_D)(2^{\lfloor n/2 \rfloor} y_E + y_D)$$

when n is even can be rewritten as

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D \tag{11}$$

and when n is odd as

$$xy = 2^{n-1} x_E y_E + 2^{\lfloor n/2 \rfloor}(x_E y_D + x_D y_E) + x_D y_D.$$

An algorithm based on this expression has a cost of

$$T(n) = 4T(n/2) + \Theta(n).$$

Karatsuba's algorithm

$$T(n) = 4T(n/2) + \Theta(n)$$

Using master theorem II, we know that $T(n) \in \Theta(n^2)$.

But if we apply Gauss's trick, we can obtain product xy doing only 3 subproducts instead of 4, hence improving the quadratic cost.

Karatsuba's algorithm

As before, we observe that

$$x_E y_D + x_D y_E = (x_E + x_D)(y_E + y_D) - x_E y_E - x_D y_D.$$

If we now call

$$a = x_E y_E, \quad b = x_D y_D, \quad c = (x_E + x_D)(y_E + y_D),$$

then the product when n is even (equation 11)

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D$$

can be rewritten as

$$2^n a + 2^{n/2}(c - a - b) + b$$

that only depends on 3 subproducts (as it happens when n is odd).

Karatsuba's algorithm

As before, we observe that

$$x_E y_D + x_D y_E = (x_E + x_D)(y_E + y_D) - x_E y_E - x_D y_D.$$

If we now call

$$a = x_E y_E, \quad b = x_D y_D, \quad c = (x_E + x_D)(y_E + y_D),$$

then the product when n is even (equation 11)

$$xy = 2^n x_E y_E + 2^{n/2}(x_E y_D + x_D y_E) + x_D y_D$$

can be rewritten as

$$2^n a + 2^{n/2}(c - a - b) + b$$

that only depends on 3 subproducts (as it happens when n is odd).

Karatsuba's algorithm

The new expression gives an algorithm of cost

$$T(n) = 3T(n/2) + \Theta(n)$$

and, due to master theorem II, we know that

$$T(n) \in \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585}).$$

Karatsuba's algorithm

Exercise 1

Explain why, for any natural number n, between n and $2n$ there is always a power of 2.

Exercise 2

Design a method for implementing Karatsuba's algorithm that does not distinguish between the even and odd case.

Explain why it has the same cost $\Theta(n^{\log_2 3})$ as Karatsuba.

Karatsuba's algorithm

Exercise 1

Explain why, for any natural number n, between n and $2n$ there is always a power of 2.

Exercise 2

Design a method for implementing Karatsuba's algorithm that does not distinguish between the even and odd case.

Explain why it has the same cost $\Theta(n^{\log_2 3})$ as Karatsuba.

Quick exponentiation

- The obvious iterative algorithm for computing x^n uses the decomposition

$$x^n = \underbrace{x \cdot x \cdot \ldots \cdot x}_{n \text{ factors}}$$

that performs $\Theta(n-1) = \Theta(n)$ multiplication.

- But with a recursive approach, we have that

$$x^n = x^{n/2} \cdot x^{n/2}$$

when n is even and

$$x^n = x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x$$

when n is odd.

Quick exponentiation

- The obvious iterative algorithm for computing x^n uses the decomposition

$$x^n = \underbrace{x \cdot x \cdot \ldots \cdot x}_{n \text{ factors}}$$

  that performs $\Theta(n-1) = \Theta(n)$ multiplication.

- But with a recursive approach, we have that

$$x^n = x^{n/2} \cdot x^{n/2}$$

  when n is even and

$$x^n = x^{(n-1)/2} \cdot x^{(n-1)/2} \cdot x$$

  when n is odd.

Quick exponentiation

Example

To compute x^{62}, the algorithm would do the following

$$x^{62} = (x^{31})^2,\ x^{31} = (x^{15})^2 \cdot x,\ x^{15} = (x^7)^2 \cdot x$$

$$x^7 = (x^3)^2 \cdot x,\ x^3 = x^2 \cdot x,\ x = 1 \cdot x$$

(in blue, the values computed recursively)

Quick exponentiation

Quick exponentiation

```
double exponential (double x, int n) {
    if (n == 0) {
        return 1;
    } else {
        double y = exponential (x, n / 2);
        if (n % 2 == 0) return y * y;
        else return y * y * x;
}   }
```

The cost computation is easy and is given by the recurrence:

$$T(n) = T(n/2) + \Theta(1)$$

that, according to master theorem II, implies

$$T(n) \in \Theta(\log n).$$

Quick exponentiation

Exercise

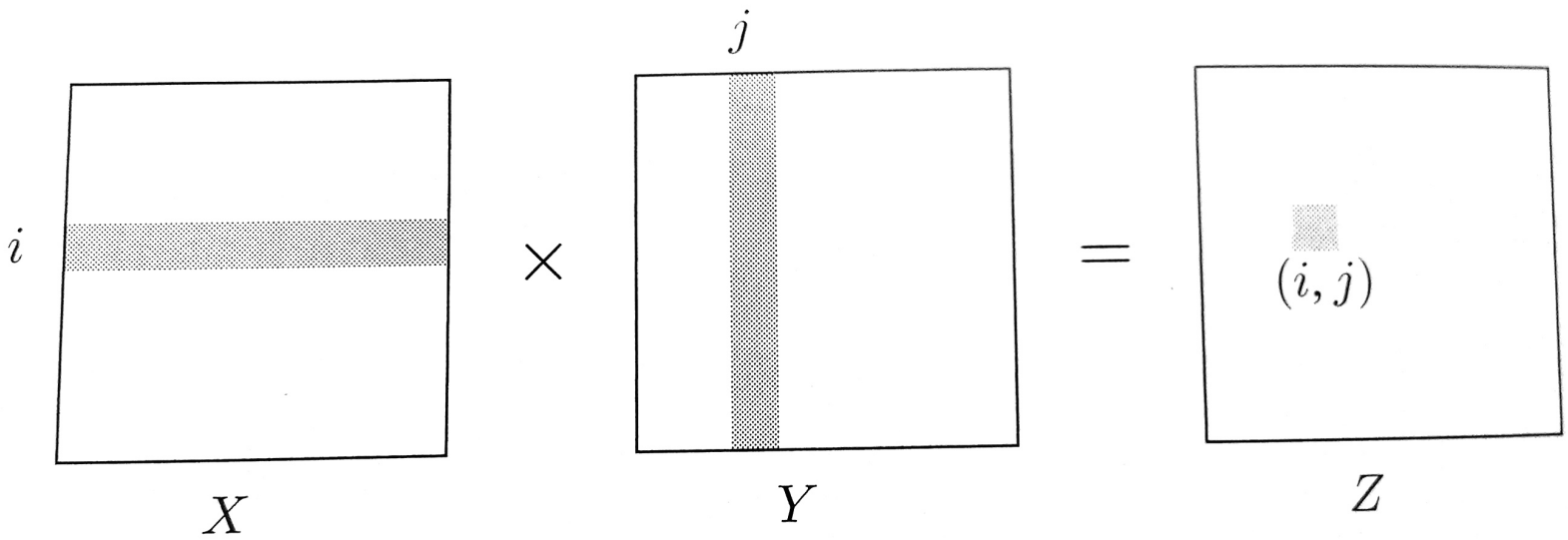
Determine whether quick exponentiation algorithm is linear or logarithmic when its cost is expressed as a function of the input size.

Strassen's algorithm

The product of two matrices $n \times n$ X i Y is a matrix $n \times n$ Z such that

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

We can think of Z_{ij} as the product of the i-th row of X by the j-th column of Y



Strassen's algorithm

Formula

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

computed for each i, j between 1 and n implies a $\Theta(n^3)$ algorithm.

It was believed that $\Theta(n^3)$ was optimal until 1969, when Volker Strassen announced an algorithm with cost $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Strassen's algorithm

Formula

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

computed for each i, j between 1 and n implies a $\Theta(n^3)$ algorithm.

It was believed that $\Theta(n^3)$ was optimal until 1969, when Volker Strassen announced an algorithm with cost $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Strassen's algorithm

Standard matrix multiplication

$\Theta(n^3)$ algorithm, adapted from *Data Structure and Alg. Analysis in C++, M.A. Weiss*.

```
matrix<int> matrix_product
          (const matrix<int> & a, const matrix<int> & b)
{
    int n = a.numrows();
    matrix<int> c(n, n);
    int i;
    for (i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            c[i][j] = 0;
    for (i = 0; i < n; j++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
     return c;
}
```

Strassen's algorithm

A first idea is that matrix multiplication can be done *by blocks*.

We split X and Y into four equal blocks each:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \; Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

It can be seen that

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Example

To compute the product AB

$$AB = \begin{bmatrix} 4 & 3 & 1 & 6 \\ 1 & 5 & 2 & 7 \\ 2 & 1 & 5 & 9 \\ 3 & 4 & 2 & 6 \end{bmatrix} \begin{bmatrix} 2 & 6 & 9 & 4 \\ 3 & 2 & 4 & 1 \\ 1 & 1 & 8 & 3 \\ 2 & 1 & 3 & 1 \end{bmatrix}$$

we define eight matrices 2×2

$$A = \begin{bmatrix} 4 & 3 \\ 1 & 5 \end{bmatrix}, B = \begin{bmatrix} 1 & 6 \\ 2 & 7 \end{bmatrix}, E = \begin{bmatrix} 2 & 6 \\ 3 & 2 \end{bmatrix}, F = \begin{bmatrix} 9 & 4 \\ 4 & 1 \end{bmatrix},$$

$$C = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}, D = \begin{bmatrix} 5 & 9 \\ 2 & 6 \end{bmatrix}, G = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}, H = \begin{bmatrix} 8 & 3 \\ 3 & 1 \end{bmatrix}.$$

Strassen's algorithm

Which is the cost of the new algorithm? Remember that it is based on

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Cost $T(n)$ is due to:

- eight matrix products of size $n/2$: $8T(n/2)$
- four sums of matrices of size $n/2$: $\Theta(n^2)$

Hence, we have

$$T(n) = 8T(n/2) + \Theta(n^2)$$

that, due to master theorem II, gives $T(n) \in \Theta(n^{\log_2 8}) = \Theta(n^3)$.

Strassen's algorithm

But the number of products can be reduced to 7.

Volker Strassen (1969)

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$P_1 = A(F - H), \; P_4 = D(G - E)$$

$$P_2 = (A + B)H, \; P_5 = (A + D)(E + H)$$

$$P_3 = (C + D)E, \; P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Using Strassen's decomposition, we obtain an algorithm with cost

$$T(n) = 7T(n/2) + \Theta(n^2)$$

that, due to master theorem II, gives $T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Strassen's algorithm

But the number of products can be reduced to 7.

Volker Strassen (1969)

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$P_1 = A(F - H), \; P_4 = D(G - E)$$

$$P_2 = (A + B)H, \quad P_5 = (A + D)(E + H)$$

$$P_3 = (C + D)E, \; P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Using Strassen's decomposition, we obtain an algorithm with cost

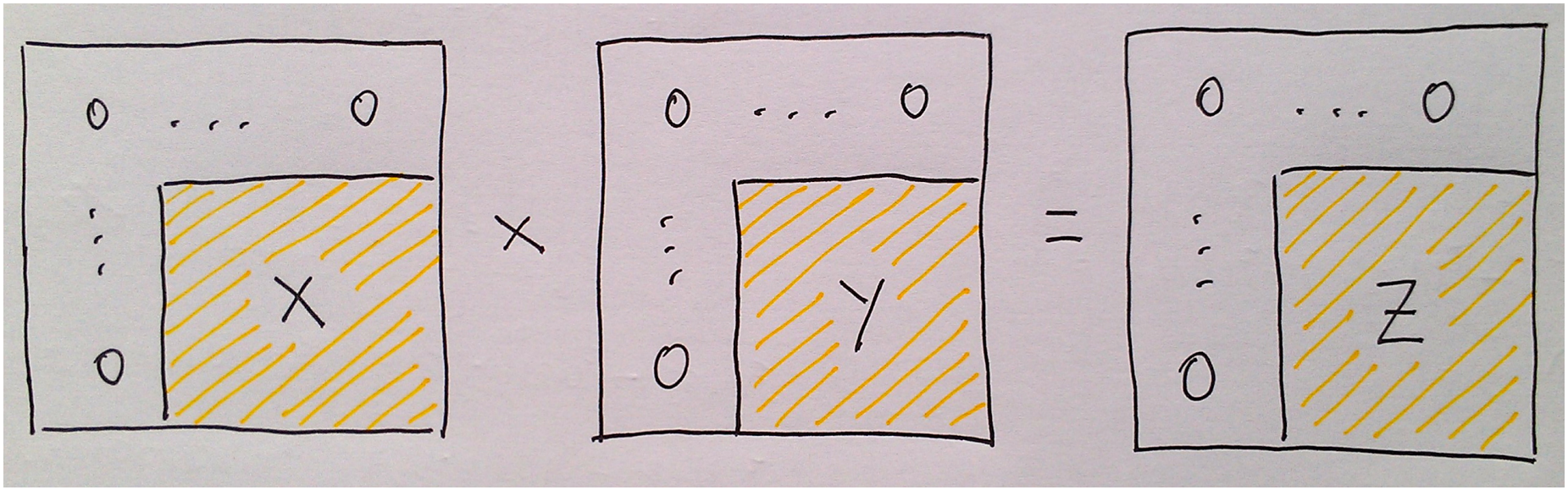
$$T(n) = 7T(n/2) + \Theta(n^2)$$

that, due to master theorem II, gives $T(n) \in \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

Strassen's algorithm

Exercise

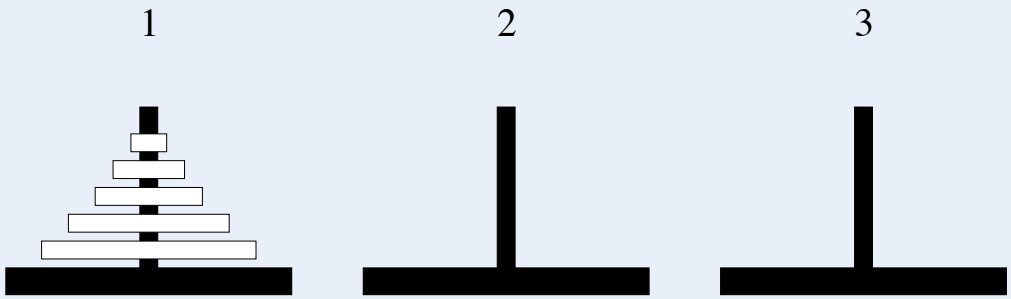
If n is not a power of 2, we generate matrices $m \times m$ where m is the first power of 2 greater than n and we insert the original matrices in them. Why doesn't the algorithm cost change?



Chapter 2. Divide-and-conquer algorithms

Towers of Hanoi

Given three rods 1, 2 and 3, and a series of disks placed in the first rod as the picture shows



the disks must be moved to rod 2 with the constraints:

- only one disk can be moved at a time
- a disk can never be placed on top of a smaller one

Towers of Hanoi

Solution 1

Imagine the rodes placed in triangular form. Now, if the movement is:

- odd, move the smallest disk to a rod clockwise
- even, do the only possible movement that does not consider the smallest disk

The problem with this solution is that it is not easy to understand.

Towers of Hanoi

Solution 1

Imagine the rodes placed in triangular form. Now, if the movement is:

- odd, move the smallest disk to a rod clockwise
- even, do the only possible movement that does not consider the smallest disk

The problem with this solution is that it is not easy to understand.

Towers of Hanoi

Solution 2

In order to move n disks from rod 1 to 2,

- if $n = 1$, move the only disk from 1 to 2.
- if $n > 1$,
  1. move $n - 1$ disks from 1 to 3
  2. move the pending disk (the largest one) from 1 to 2
  3. move $n - 1$ disks from 3 to 2

Towers of Hanoi

Algorithm

```
void hanoi(int n, int a, int b, int c){
    if (n > 0) {
        hanoi(n-1, a, c, b);
        cout << a, b;
        hanoi(n-1, c, b, a);
}   }
```

Cost

The cost corresponds to the number of movements. We define

$$T(n) = \text{ number of movements of } \texttt{hanoi(n,1,2,3)}.$$

Then,

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1, \quad \text{if } n > 0.$$

Towers of Hanoi

Recurrence

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1, \quad \text{if } n > 0.$$

Asymptotic solution

Applying master theorem I, we get $T \in \Theta(2^n)$.

Exact solution

We define $S(n) = T(n) + 1$ and we write it independently of $T(n)$:

$$S(0) = 1$$

$$S(n) = 2T(n-1) + 2 = 2(S(n-1) - 1) + 2 = 2S(n-1), \quad \text{if } n > 0.$$

Now, $S(n)$ can be solved directly and we get: $S(n) = 2^n$ for all $n \geq 0$.
Hence

$$T(n) = 2^n - 1 \text{ for all } n \geq 0.$$

Towers of Hanoi

Recurrence

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1, \quad \text{if } n > 0.$$

Asymptotic solution

Applying master theorem I, we get $T \in \Theta(2^n)$.

Exact solution

We define $S(n) = T(n) + 1$ and we write it independently of $T(n)$:

$$S(0) = 1$$

$$S(n) = 2T(n-1) + 2 = 2(S(n-1) - 1) + 2 = 2S(n-1), \quad \text{if } n > 0.$$

Now, $S(n)$ can be solved directly and we get: $S(n) = 2^n$ for all $n \geq 0$.
Hence

$$T(n) = 2^n - 1 \text{ for all } n \geq 0.$$

Towers of Hanoi

Recurrence

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1, \quad \text{if } n > 0.$$

Asymptotic solution

Applying master theorem I, we get $T \in \Theta(2^n)$.

Exact solution

We define $S(n) = T(n) + 1$ and we write it independently of $T(n)$:

$$S(0) = 1$$

$$S(n) = 2T(n-1) + 2 = 2(S(n-1) - 1) + 2 = 2S(n-1), \quad \text{if } n > 0.$$

Now, $S(n)$ can be solved directly and we get: $S(n) = 2^n$ for all $n \geq 0$.
Hence

$$T(n) = 2^n - 1 \text{ for all } n \geq 0.$$

Median

Definition

The median of a list of numbers is the element for which there is the same amount of smaller and larger numbers.

For example, the median of $(15, 3, 34, 5, 10)$ is 10 because if we write them in order it is the one in the middle:

$$3 \;\; 5 \;\; 10 \;\; 15 \;\; 34$$

If the list has odd size, we have two possibilities. We pick, for example, the smallest one.

Median

Characteristics of the median:

- It is always one of the values of the data set
- It is less sensitive to outliers. For example:
  1. Given numbers 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 100 (10 times 1 and once 100),
     - the average is 10
     - the median is 1
  2. Given 2, 4, 6, 8, 10.000,
     - the average is 1002
     - the median is 6

Median

To compute the median, it is enough to sort the elements.

The problem is that sorting takes $\Theta(n \log n)$ time

$$[8, 0, 3, 10, 5, 7, 12, 3, 7, 2, 9, 1, 6]$$

$$\Downarrow$$

$$[0, 1, 2, 3, 3, 5, 6, 7, 8, 9, 10, 12]$$

But this does more work than necessary:

we only want the central element. No need to sort the rest

$$\{8, 0, 3, 10, 5, 7\}, 6, \{12, 3, 7, 2, 9, 1\}$$

Median

To compute the median, it is enough to sort the elements.

The problem is that sorting takes $\Theta(n \log n)$ time

$$[8, 0, 3, 10, 5, 7, 12, 3, 7, 2, 9, 1, 6]$$

$$\Downarrow$$

$$[0, 1, 2, 3, 3, 5, 6, 7, 8, 9, 10, 12]$$

But this does more work than necessary:

we only want the central element. No need to sort the rest

$$\{8, 0, 3, 10, 5, 7\}, 6, \{12, 3, 7, 2, 9, 1\}$$

Median

It is sometimes easier to work with a more general version of the problem. Here, we choose the selection problem.

Definition

If S is a list and $k \geq 1$, we call

$$\text{selection}(S, k)$$

to the k-th smallest element of S.

Selection problem

Given a list S and a natural number k, determine $\text{selection}(S, k)$.

The median of a list S with n numbers is $\text{selection}(S, \lfloor n/2 \rfloor)$.

Median

It is sometimes easier to work with a more general version of the problem. Here, we choose the selection problem.

Definition

If S is a list and $k \geq 1$, we call

$$\text{selection}(S, k)$$

to the k-th smallest element of S.

Selection problem

Given a list S and a natural number k, determine selection(S, k).

The median of a list S with n numbers is selection$(S, \lfloor n/2 \rfloor)$.

Median

Selection problem

Given a list S and a natural number k, determine selection(S, k).

Idea for an algorithm

For each number x, divide the list in 3 parts:

- elements smaller than x
- elements equal to x
- elements larger than x

If we have the vector

$$S : \left[\begin{array}{cccccccccccc} 2 & 36 & 5 & 21 & 8 & 13 & 11 & 20 & 5 & 4 & 1 \end{array}\right]$$

for $x = 5$ we divide it into

$$S_E : \left[\begin{array}{ccc} 2 & 4 & 1 \end{array}\right] \quad S_x : \left[\begin{array}{cc} 5 & 5 \end{array}\right] \quad S_D : \left[\begin{array}{cccccc} 36 & 21 & 8 & 13 & 11 & 20 \end{array}\right]$$

Median

Idea for an algorithm

Assume that want the 8th element of S

$$S : \left[\begin{array}{cccccccccccc} 2 & 36 & 5 & 21 & 8 & 13 & 11 & 20 & 5 & 4 & 1 \end{array}\right]$$

We know that it will be the 3rd element of S_D because $|S_E| + |S_x| = 5$.

$$S_E : \left[\begin{array}{ccc} 2 & 4 & 1 \end{array}\right] \quad S_x : \left[\begin{array}{cc} 5 & 5 \end{array}\right] \quad S_D : \left[\begin{array}{cccccc} 36 & 21 & 8 & 13 & 11 & 20 \end{array}\right]$$

We can define the operator selection(S, k) recursively:

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_E, k), & \text{if } k \leq |S_E| \\ x, & \text{if } |S_E| < k \leq |S_E| + |S_x| \\ \text{selection}(S_D, k - |S_E| - |S_x|), & \text{if } k > |S_E| + |S_x| \end{cases}$$

Idea for an algorithm

Assume that want the 8th element of S

$$S : \left[\begin{array}{cccccccccc} 2 & 36 & 5 & 21 & 8 & 13 & 11 & 20 & 5 & 4 & 1 \end{array}\right]$$

We know that it will be the 3rd element of S_D because $|S_E| + |S_x| = 5$.

$$S_E : \left[\begin{array}{ccc} 2 & 4 & 1 \end{array}\right] \quad S_x : \left[\begin{array}{cc} 5 & 5 \end{array}\right] \quad S_D : \left[\begin{array}{cccccc} 36 & 21 & 8 & 13 & 11 & 20 \end{array}\right]$$

We can define the operator selection(S, k) recursively:

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_E, k), & \text{if } k \leq |S_E| \\ x, & \text{if } |S_E| < k \leq |S_E| + |S_x| \\ \text{selection}(S_D, k - |S_E| - |S_x|), & \text{if } k > |S_E| + |S_x| \end{cases}$$

Median

Sketch of the algorithm

$$\text{selection}(S,k) = \begin{cases} \text{selection}(S_E,k), & \text{if } k \leq |S_E| \\ x, & \text{if } |S_E| < k \leq |S_E| + |S_x| \\ \text{selection}(S_D, k - |S_E| - |S_x|), & \text{if } k > |S_E| + |S_x| \end{cases}$$

Algorithm:

1. Choose an element x at random
2. Divide S with respect to x: S_E, S_x, S_D
3. Compute $\text{selection}(S,k)$ recursively

On average, the second chose element is in the range 25%–75% and hence, it will have reduced the problem in $3/4$ parts. This gives

$$T(n) \leq T(3n/4) + \mathcal{O}(n)$$

which implies that $T(n) \in \mathcal{O}(n)$ on average.