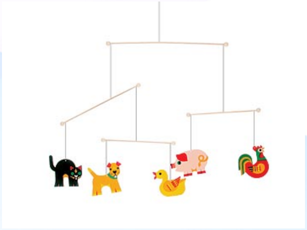
Lecture Notes on Data Structures and Algorithms: Divide and Conquer Algorithms

Conrado Martínez
U. Politècnica Catalunya

April 1, 2016



Part II

Divide and Conquer Algorithms

Introduction

The basic principle of divide and conquer (in Catalan, *dividir per conquerir*, in Spanish *divide y vencerás*) is very simple:

1. If the given input is simple enough, find the corresponding solution using some direct straightforward method.
2. Otherwise *divide* the given input x into subinstances $x_1, \ldots, x_k$ of the problem, and solve it, independently and recursively, for each of the subinstances.
3. The solutions thus obtained $y_1, \ldots, y_k$ are *combined* to get the solution corresponding to the original input x.

Divide and Conquer: A Template

- The template for divide and conquer algorithms in pseudocode looks like:

```
procedure DIVIDE_AND_CONQUER($x$)
    if $x$ is simple then
        return DIRECT_SOLUTION($x$)
    else
        $\langle x_1, x_2, \ldots, x_k \rangle$ := DIVIDE($x$)
        for :=1 to $k$ do
            $y_i$ := DIVIDE_AND_CONQUER($x_i$)
        end for
        return COMBINE($y_1, y_2, \ldots, y_k$)
    end if
end procedure
```

- Sometimes, when $k = 1$, the algorithm is called a *reduction algorithm*.

Divide and Conquer: Properties

Divide and conquer algorithms share some common traits:

- No subinstance of the problem is solved more than once
- The size of the subinstances is, at least on average, a fraction of the size of the original input; that is, if x is of size n then the average size of any x_i is n/c_i for some $c_i > 1$. Very often the condition holds not only on average, but surely.

Cost of Divide and Conquer Algorithms

For many Divide and Conquer algorithms the cost satisfies a recurrence of the form

$$T(n) = \begin{cases} f(n) + a \cdot T(n/b) & \text{if } n > n_0, \\ g(n) & \text{if } n \le n_0. \end{cases}$$

This is the case when a direct solution is returned whenever the size of the input is $n \le n_0$ for some n_0, and otherwise the original input is divided into subinstances of size (roughly) n/b each, and a recursive calls are made. The cost of computing the direct solution is $g(n)$, and the cost of the *divide* and *combine* steps is $f(n)$. In most cases, $a = b$, but there are also many examples of divide and conquer algorithms where this is not true.

Cost of Divide and Conquer Algorithms

Recall:

Theorem

If $f(n) = \Theta(n^k)$ *and*

$$T(n) = \begin{cases} f(n) + a \cdot T(n/b) & \text{if } n > n_0, \\ g(n) & \text{if } n \leq n_0, \end{cases}$$

with constants $a \geq 1$ *and* $b > 1$, *then*

$$T(n) = \begin{cases} \Theta(f(n)) & \text{if } \alpha < k, \\ \Theta(f(n) \log n) & \text{if } \alpha = k \\ \Theta(n^\alpha) & \text{if } \alpha > k, \end{cases}$$

where $\alpha = \log_b a$.

Cost of Divide and Conquer Algorithms

Proof (sketch): We will assume that $n = n_0 \cdot b^j$. By repeatedly applying the recurrence

$$\begin{aligned}
T(n) &= f(n) + a \cdot T(n/b) \\
&= f(n) + a \cdot f(n/b) + a^2 \cdot T(n/b^2) \\
&= \cdots \\
&= f(n) + a \cdot f(n/b) + \cdots + a^{j-1} \cdot f(n/b^{j-1}) \\
&\qquad + a^j \cdot T(n/b^j) \\
&= \sum_{0 \le i < j} a^i \cdot f(n/b^i) + a^j \cdot T(n_0) \\
&= \Theta\left(\sum_{0 \le i < j} a^i \left(\frac{n}{b^i} \right)^k \right) + a^j \cdot g(n_0) \\
&= \Theta\left(n^k \cdot \sum_{0 \le i < j} \left(\frac{a}{b^k} \right)^i \right) + a^j \cdot g(n_0).
\end{aligned}$$

Cost of Divide and Conquer Algorithms

Since $g(n_0)$ is a constant, the second term is $\Theta(n^\alpha)$:

$$\begin{aligned} a^j &= a^{\log_b(n/n_0)} = a^{\log_b n} \cdot a^{-\log_b n_0} \\ &= \Theta(b^{\log_b a \cdot \log_b n}) \\ &= \Theta(n^{\log_b a}) = \Theta(n^\alpha). \end{aligned}$$

We have thus three different cases to consider, according to a/b^k being less, equal or greater than 1. Equivalently, since $\alpha = \log_b a$, the three cases correspond to α being greater, equal or less than k. If $k > \alpha$ ($\equiv a/b^k < 1$) then the sum of the first term is a geometric series with ratio less than 1 and its value is upper bounded by a constant; therefore the first term is $\Theta(n^k)$. Since $k > \alpha$, the first term is dominant and $T(n) = \Theta(n^k) = \Theta(f(n))$.

Cost of Divide and Conquer Algorithms

If $\alpha = k$ then $a/b^k = 1$ and the sum adds to j; as $j = \log_b(n/n_0) = \Theta(\log n)$, we conclude that $T(n) = \Theta(n^k \cdot \log n) = \Theta(f(n) \cdot \log n)$. Last, but not least, if $k < \alpha$ then $a/b^k > 1$ and the sum is

$$\sum_{0 \le i < j} \left(\frac{a}{b^k}\right)^i = \frac{(a/b^k)^j - 1}{a/b^k - 1} = \Theta\left(\left(\frac{a}{b^k}\right)^j\right)$$
$$= \Theta\left(\frac{n^\alpha}{n^k}\right) = \Theta(n^{\alpha - k}).$$

In this case the second term dominates and $T(n) = \Theta(n^\alpha)$.

Part II

Divide and Conquer Algorithms

Part II

Divide and Conquer Algorithms

Karatsuba's Algorithm

The traditional algorithm for integer multiplication has cost $\Theta(n^2)$ to multiply two integers of n bits each, since it is equivalent to performing n additions of $\Theta(n)$-bit numbers and each such addition has cost $\Theta(n)$.

```
231 x 659 = 9 x 231 + 5 x 231 x 10
          + 6 x 231 x 100
          = 2079 + 11550 + 138600
          = 152229
```

Karatsuba's Algorithm

The Russian multiplication algorithm has quadratic cost too.

```
$z := 0$
▷ $x = X \wedge y = Y$
**while** $x \neq 0$ **do**
    ▷ Inv: $z + x \cdot y = X \cdot Y$
    **if** $x$ is even **then**
        $x := x$ **div** $2; y := 2 \cdot y$
    **else**
        $x := x - 1; z := z + y$
    **end if**
**end while**
▷ $z = X \cdot Y$
```

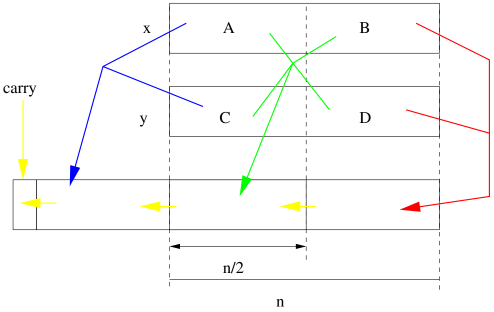
Karatsuba's Algorithm

```
// Pre: $x = X \wedge y = Y$
z = 0;
while (x != 0)
// Inv: $z + x \cdot y = X \cdot Y$
  if (x % 2 == 0) { x /= 2; y *= 2; }
  else { x--; z += y; }
// Post: $z = X \cdot Y$
```

Karatsuba's Algorithm

Assume both x and y are n-bit integers for $n = 2^k$ (if n is not a power of 2 or they are not both of n bits, they can be padded with 0s to achieve this).

An idea that does not work:



$$x \cdot y = (A \cdot 2^{n/2} + B) \cdot (C \cdot 2^{n/2} + D)$$
$$= A \cdot C \cdot 2^n + (A \cdot D + B \cdot C) \cdot 2^{n/2} + B \cdot D.$$

Karatsuba's Algorithm

Once we decompose x as $x = \langle A, B\rangle$ and $y = \langle C, D\rangle$, we compute the 4 products $A \cdot B$, $A \cdot D$, $B \cdot C$ and $B \cdot D$, and then combine these intermediate results by means of sums and *shifts*. The cost of the sums and the shifts is $\Theta(n)$. Hence, the cost of this divide-and-conquer multiplication algorithm satisfies the recurrence

$$M(n) = \Theta(n) + 4 \cdot M(n/2), \quad n > 1$$

and the solution is $M(n) = \Theta(n^2)$, since $k = 1$, $a = 4$ and $b = 2$ ($k = 1 < \alpha = \log_2 4 = 2$).

Karatsuba's Algorithm

Karatsuba's algorithm (1962) computes the product of x and y dividing the two numbers as before but only 3 products need to be recursively computed:

$$\begin{aligned} U &= A \cdot C \\ V &= B \cdot D \\ W &= (A + B) \cdot (C + D) \end{aligned}$$

The final result is obtained combining the intermediate results above as follows:

$$x \cdot y = U \cdot 2^n + (W - (U + V)) \cdot 2^{n/2} + V.$$

Karatsuba's Algorithm

The algorithm requires 6 sums (one is a substraction) instead of the 3 sums needed in the previous algorithm. But the non-recursive cost of Karatsuba's algorithm is still linear, and since the number of recursive calls is smaller, the overall cost is asymptotically faster:

$$M(n) = \Theta(n) + 3 \cdot M(n/2)$$

$$M(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.5849625\ldots})$$

The hidden constant in $\Theta(n^{1.5849625\ldots})$ is large and the advantage of Karatsuba's algorithm w.r.t. to the basic quadratic algorithm is not noticeable unless n is large enough (typically we see no improvement until n is about 200 to 250 bits)

Karatsuba's Algorithm

```
  procedure MULT($x$,$y$, $i$, $j$, $i'$, $j'$)
Require: $1 \le i \le j \le n$, $1 \le i' \le j' \le n$, $j' - i' = j - i$
Ensure: Returns the product of $x[i..j]$ and $y[i'..j']$
    $n := j - i + 1$
    if $n < M$ then
        Use a basic algorithm (hardware?) to compute the product
    else
        $m := (i + j)$ div $2$; $m' := (i' + j')$ div $2$
        $U :=$ MULT$(x, y, m + 1, j, m' + 1, j')$
        $V :=$ MULT$(x, y, i, m, i', m')$
        $U$.LEFT_SHIFT$(n)$
        $W_1 := x[i..m] + y[i'..m']$; $W_2 := x[m + 1..j] + y[m' + 1..j']$
        Pad $W_1$ or $W_2$ if necessary
        $W :=$ MULT$(W_1, W_2, \ldots)$
        $W := W - (U + V)$
        $W$.LEFT_SHIFT$(n$ div $2)$
        return $U + W + V$
    end if
  end procedure
```

Part II

Divide and Conquer Algorithms

Strassen's Algorithm

The multiplication of two square matrices $n \times n$ has cost $\Theta(n^3)$ using the straightforward definition of matrix product:

```
for $i := 1$ to $n$ do
    for $j := 1$ to $n$ do
        $C[i, j] := 0$
        for $k := 1$ to $n$ do
            $C[i, j] := C[i, j] + A[i, k] * B[k, j]$
        end for
    end for
end for
```

Strassen's Algorithm

To apply the divide-and-conquer scheme here, we divide the original matrices into four blocks each (a block is a submatrix $\frac{n}{2} \times \frac{n}{2}$)

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \cdot \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right)$$

The rule for matrix multiplication carries over to blocks: thus $C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$, etc.

Strassen’s Algorithm

Each block in C requires two block multiplications and a sum (the cost of the sum is $\Theta(n^2)$). The cost of a naïve divide-and-conquer algorithm is still cubic, since the recurrence is

$$M(n) = \Theta(n^2) + 8 \cdot M(n/2),$$

and the solution is $M(n) = \Theta(n^3)$.
To achieve a more efficient solution we need to reduce the number of recursive calls, like in Karatsuba’s algorithm.

Strassen's Algorithm

Strassen (1969) proposed such a way[1]. One of the things that make this derivation long and complicated is the fact that matrix product is not commutative.
We have to obtain the following 7 blocks (matrices $n/2 \times n/2$), using 7 products (recursively) and 14 matrix additions/substractions

$$
\begin{aligned}
M_1 &= (A_{21} + A_{22} - A_{11}) \cdot (B_{11} + B_{22} - B_{12}) \\
M_2 &= A_{11} \cdot B_{11} \\
M_3 &= A_{12} \cdot B_{21} \\
M_4 &= (A_{11} - A_{21}) \cdot (B_{22} - B_{12}) \\
M_5 &= (A_{21} + A_{22}) \cdot (B_{12} - B_{11}) \\
M_6 &= (A_{12} - A_{21} + A_{11} - A_{22}) \cdot B_{22} \\
M_7 &= A_{22} \cdot (B_{11} + B_{22} - B_{12} - B_{21})
\end{aligned}
$$

[1] Brassard & Bratley's *Fundamentals of Algorithmics* give a detailed description of how to find the set of formulas behind Strassen's algorithm.

Strassen's Algorithm

With 10 matrix additions/substractions more, we get the blocks of C:

$$\begin{aligned} C_{11} &= M_2 + M_3 \\ C_{12} &= M_1 + M_2 + M_5 + M_6 \\ C_{21} &= M_1 + M_2 + M_4 - M_7 \\ C_{22} &= M_1 + M_2 + M_4 + M_5 \end{aligned}$$

Since the matrix additions/substractions have cost $\Theta(n^2)$, the cost of Strassen's algorithm is given by the recurrence:

$$M(n) = \Theta(n^2) + 7 \cdot M(n/2),$$

where the solution is $\Theta(n^{\log_2 7}) = \Theta(n^{2.807\ldots})$.

Strassen’s Algorithm

Strassen’s algorithm had an enourmous impact among theoreticians as it was the first matrix multiplication algorithm with complexity asymptotically smaller than n^3; that also had implications for the development of more efficient algorithms to compute inverse matrices and determinants, solving linear systems, etc.

Strassen’s algorithm was one of the first examples in which novel algorithmic techniques helped to break a seemingly unbreakable barrier. Other examples include Karatsuba’s algorithm and the Fast Fourier Transform (FFT), which also belong to the family of divide-and-conquer algorithms.

Strassen's Algorithm

In the years following Strassen's breakthrough, more efficient matrix multiplication algorithms have been developed. The current record is held by the theoretical superior but impractical algorithm of Coppersmith and Winograd (1986) with cost $\Theta(n^{2.376\ldots})$.

Strassen's algorithm is not faster in practice than the ordinary matrix multiplication algorithm, unless n is rather large ($n \gg 500$), since the hidden constants and lower order terms in the cost are quite big and their impact in the cost cannot be disregarded except when n is large enough.

Part II

Divide and Conquer Algorithms

Part II

Divide and Conquer Algorithms

Binary Search

Given an array $A[1..n]$ with n elements, in increasing order, and some element x, find the value i, $0 \leq i \leq n$, such that $A[i] \leq x < A[i+1]$ (with the convention that $A[0] = -\infty$ and $A[n+1] = +\infty$).
If the array were empty the answer is simple, since x is not in the array. But it is clear now that we must work with a generalization of the original problem: given a segment or subarray $A[\ell+1..u-1]$, $0 \leq \ell \leq u \leq n+1$, in increasing order, and such that $A[\ell] \leq x < A[u]$, find the value i, $\ell \leq i \leq u-1$, such that $A[i] \leq x < A[i+1]$ (this is a typical example of *parameter embedding*).
For the generalization, if $\ell + 1 = u$, the segment is empty and we can return $i = u - 1 = \ell$.

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Binary Search

Example

We are looking for $x = 88$

10	23	37	41	52	66	78	83	90	101	115	124	136	147	159

Search ends at position $k = 8$: $A[k] = 83 \leq x < A[k+1] = 90$

Binary Search

```
  ▷ Initial call: BinSearch$(A, x, 0, n+1)$
  procedure BinSearch($A$, $x$, $\ell$, $u$)
Require: $0 \le \ell < u \le n+1$, $A[\ell] \le x < A[u]$
Ensure: Return $i$ such that $A[i] \le x < A[i+1]$ and $\ell \le i < u$
      if $\ell + 1 = u$ then
          return $\ell$
      else
          $m := (\ell + u)$ div $2$
          if $x < A[m]$ then
              return BinSearch$(A, x, \ell, m)$
          else
              return BinSearch$(A, x, m, u)$
          end if
      end if
  end procedure
```

Binary Search

```
// Initial call: int p = bsearch(A, x, -1, A.size());

template <class T>
int bsearch(vector<T>& A, const T& x,
            int l, int u) {
    if (l == u + 1)
        return u;
    int m = (l + u) / 2;
    if (x < A[m])
       return bsearch(A, x, l, m);
    else
       return bsearch(A, x, m, u);
}
```

Binary Search

Let n denote the size of the array A where we perform the search. Neglecting the small difference between the actual size of the subarray in the recursive call and $n/2$, the cost of a binary search satisfies the recurrence

$$B(n) = \Theta(1) + B(n/2), \quad n > 0,$$

and $B(0) = b_0$, since each call makes one recursive call, either in the “left” or the “right”, half of the current segment. Using Theorem 1, $k = 0$ and $\alpha = \log_2 1 = 0$ and thus $B(n) = \Theta(\log n)$.

Part II

Divide and Conquer Algorithms

Mergesort

Mergesort is one of the oldest efficient sorting methods ever proposed. Different variants of this method are particularly useful for external memory sorts. Mergesort is also one of the best sorting methods for linked lists.

The basic idea is simple: divide the input sequence in two halves, sort each half recursively and obtain the final result *merging* the two sorted sequences of the previous step.

Mergesort

We will assume here that the input is a simple linked list L that stores the sequence of elements $x_1, \ldots, x_n$. Each element is stored in a node of the linked list with two fields: `info` stores the element and `next` is a pointer to the next node in the list (the special value `next` = **null**indicates that the node is the last in the list). The list L is, in fact, a pointer to its first node. By *p* $\rightarrow$ *f* we mean the field f in the object pointed to by pointer p, like in C or C++.

Mergesort

```
  procedure SPLIT($L$, $L'$, $n$)
Require: $L = [\ell_1, \dots, \ell_m], m \geq n$
Ensure: $L = [\ell_1, \dots, \ell_n], L' = [\ell_{n+1}, \dots, \ell_m]$
      $p := L$
      while $n > 1$ do
          $p := p \rightarrow$ next
          $n := n - 1$
      end while
      $L' := p \rightarrow$ next; $p \rightarrow$ next $:=$ null
  end procedure
```

Mergesort

```
procedure MergeSort($L$, $n$)
    if $n > 1$ then
        $m := n$ div $2$
        Split($L, L', m$)
        MergeSort($L, m$)
        MergeSort($L', n - m$)
        ▷ merge the two lists $L$ and $L'$
        $L :=$ Merge($L$, $L'$)
    end if
end procedure
```

Mergesort

```
struct node_list {
       Elem info;
       node_list* next;
};
typedef node_list* List;

void split(List& L, List& L2, int n) {
  node_list* p = L;
  while (n > 1) {
    p = p -> next;
    --n;
  }
  L2 = p -> next; p -> next = NULL;
}

List merge(List& L1, List& L2);

void mergesort(List& L, int n) {
  if (n > 1) {
    int m = n / 2;
    List L2;
    split(L, L2, m);
    mergesort(L, m);
    mergesort(L2, n-m);
    L = merge(L, L2);
  }
}
```

Mergesort

To develop the `Merge` procedure we reason as follows: if L or L' is empty the result of the merging is the other list, that is, the one which is (usually) not empty. When both lists are non-empty, we compare their respective first elements. The smaller of the two must be the first element in the merged list, and the remaining elements are the result of merging (recursively) the tail of the list containing the smallest element with the other list.

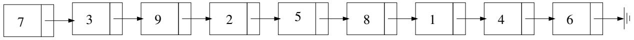
Mergesort

```
procedure MERGE($L$, $L'$)
    if $L$ = null then return $L'$
    end if
    if $L'$ = null then return $L$
    end if
    if $L \to$ info $\leq L' \to$ info then
        $L \to$ next := MERGE($L \to$ next, $L'$)
        return $L$
    else
        $L' \to$ next := MERGE($L, L' \to$ next)
        return $L'$
    end if
end procedure
```

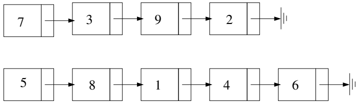
Mergesort

```
List merge(List L, List L2) {
     if (L == NULL) return L2;
     if (L2 == NULL) return L;
     if (L -> info <= L2 -> info) {
        L -> next = merge(L -> next, L2);
        return L;
     } else { // L -> info > L2 -> info
        L2 -> next = merge(L, L2 -> next);
        return L2;
     }
}
```

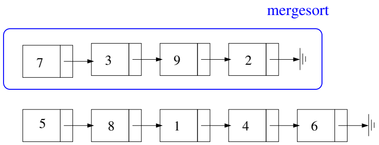
Mergesort



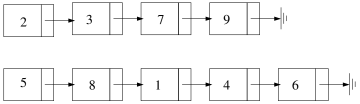
Mergesort



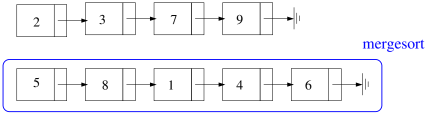
Mergesort



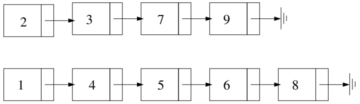
Mergesort



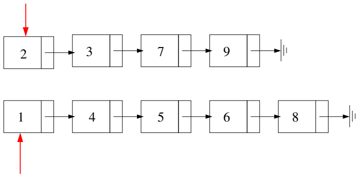
Mergesort



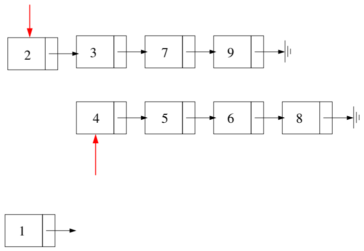
Mergesort



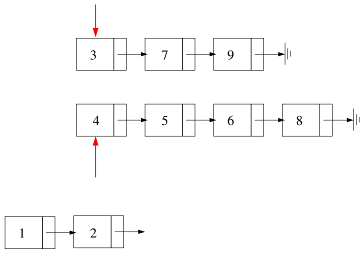
Mergesort



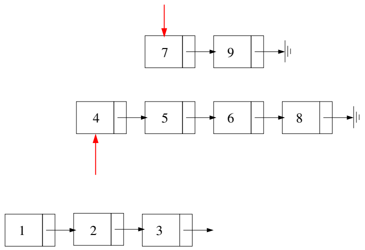
Mergesort



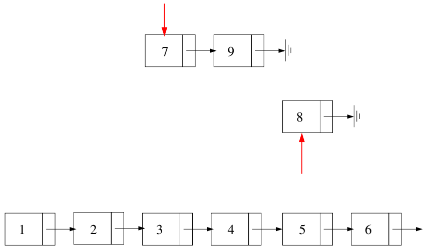
Mergesort



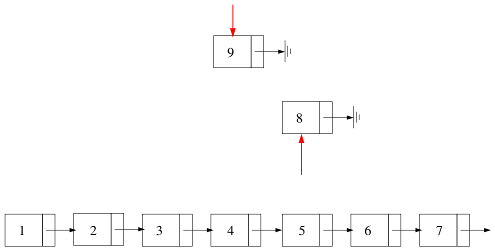
Mergesort



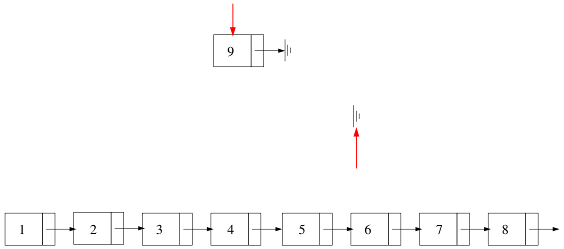
Mergesort



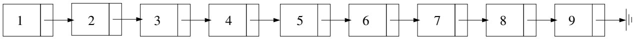
Mergesort



Mergesort



Mergesort



Mergesort

Each element of L and L' is "visited" exactly once, hence the cost of MERGE is proportional to the sum of the lengths of the lists L and L'; that is, the cost is $\Theta(n)$. Since MERGE has *tail recursion* it is quite easy to obtain an iterative version which is slightly more efficient.
Note that if the two inspected elements in L and L' are identical, we put first the element coming from L into the result, which ensures that MERGESORT is a *stable* sorting method.
The cost of MERGESORT is described by the recurrence:

$$\begin{aligned} M(n) &= \Theta(n) + M\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + M\left(\left\lceil \frac{n}{2} \right\rceil\right) \\ &= \Theta(n) + 2 \cdot M\left(\frac{n}{2}\right) \end{aligned}$$

hence the solution is $M(n) = \Theta(n \log n)$, by applying the second case of Theorem 1.

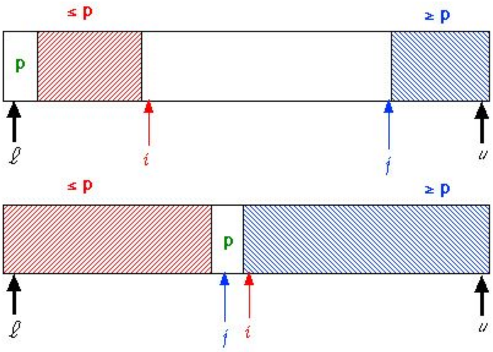
Part II

Divide and Conquer Algorithms

QuickSort

QUICKSORT (Hoare, 1962) is a sorting algorithm using the divide-and-conquer principle too, but contrary to the previous examples, it does not guarantee that the size of each subinstance will be a fraction of the size of the original given instance.

The basis of *Quicksort* is the procedure PARTITION: given an element p, called the pivot, the (sub)array is rearranged as shown in the picture below.



QuickSort

PARTITION puts the pivot in its final place. Hence it suffices to recursively sort the subarrays to its left and to its right. While *divide* is simple and *combine* does most of the work in MERGESORT, in QUICKSORT it happens just the contrary: *divide* consists in partitioning the array, and we have nothing to do to *combine* the solutions obtained from the recursive calls. The QUICKSORT algorithm for a subarray $A[\ell..u]$ is

```
  procedure QUICKSORT($A$, $\ell$, $u$)
Ensure: Sorts subarray $A[\ell..u]$
    if $u - \ell + 1 \leq M$ then
        use a simple sorting method, e.g., insertion sort
    else
        PARTITION$(A, \ell, u, k)$
        $\triangleright$ $A[\ell..k-1] \leq A[k] \leq A[k+1..u]$
        QUICKSORT$((A, \ell, k-1)$
        QUICKSORT$(A, k+1, u)$
    end if
  end procedure
```

QuickSort

Instead of using a simple sorting method each time we reach a subarray with M or less elements, we can postpone the sorting of small subarrays until the end:

```
QUICKSORT($A, 1, A$.SIZE())
INSERTSORT($A, 1, A$.SIZE())
```

Since the array is almost sorted upon completion of the call to QUICKSORT, the last step using INSERTSORT needs only linear time ($\Theta(n)$), where $n = A.\text{SIZE}()$.

The typical optimal choice for the cut-off value M is around 20 to 25.

QuickSort

```
template <class T>
void partition(vector<T>& v, int l, int u, int& k);

template <class T>
void quicksort(vector<T>& v, int l, int u) {
     if (u -l + 1 > M) {
        int k;
        partition(v, l, u, k);
        quicksort(v, l, k-1);
        quicksort(v, k+1, u);
     }
}

template <class T>
void quicksort(vector<T>& v) {
     quicksort(v, 0, v.size()-1);
     insertion_sort(v, 0, v.size()-1);
}
```

QuickSort

There are many ways to do the partition; not them all are equally good. Some issues, like repeated elements, have to be dealt with carefully. Bentley & McIlroy (1993) discuss a very efficient partition procedure, which works seamlessly even in the presence of repeated elements. Here, we will examine a basic algorithm, which is reasonably efficient.
We will keep two indices i and j such that $A[\ell + 1..i - 1]$ contains elements less than or equal to the pivot p, and $A[j + 1..u]$ contains elements greater than or equal to the pivot p. The two indices scan the subarray locations, i from left to right, j from right to left, until $A[i] > p$ and $A[j] < p$, or until they cross ($i = j + 1$).

QuickSort

```
  procedure PARTITION($A$, $\ell$, $u$, $k$)
Require: $\ell \leq u$
Ensure: $A[\ell..k-1] \leq A[k] \leq A[k+1..u]$
      $i := \ell + 1; j := u; p := A[\ell]$
      while $i < j + 1$ do
          while $i < j + 1 \land A[i] \leq p$ do
              $i := i + 1$
          end while
          while $i < j + 1 \land A[j] \geq p$ do
              $j := j - 1$
          end while
          if $i < j + 1$ then
              $A[i] :=: A[j]$
              $i := i + 1; j := j - 1$
          end if
      end while
      $A[\ell] :=: A[j]; k := j$
  end procedure
```

QuickSort

```
template <class T>
void partition(vector<T>& v, int l, int u, int& k) {
      int i = l;
      int j = u + 1;
      T pv = v[i]; // simple choice for pivot
      do {
           while (v[i] <  pv and i < u) ++i;
           while (pv < v[j] and l < j) --j;
           if (i <= j) {
             swap(v[i], v[j]);
             ++i; --j;
           }
      } while (i <= j);
      swap(v[l], v[j]);
      k = j;
}
```

The Cost of QuickSort

The worst-case cost of QUICKSORT is $\Theta(n^2)$, hence not very attractive. But it only occurs is all or most recursive call one of the subarrays contains very few elements and the other contains almost all. That would happen if we systematically choose the first element of the current subarray as the pivot and the array is already sorted!
The cost of the partition is $\Theta(n)$ and we would have then

$$\begin{aligned} Q(n) &= \Theta(n) + Q(n-1) + Q(0) \\ &= \Theta(n) + Q(n-1) = \Theta(n) + \Theta(n-1) + Q(n-2) \\ &= \cdots = \sum_{i=0}^{n} \Theta(i) = \Theta\left(\sum_{0 \le i \le n} i\right) \\ &= \Theta(n^2). \end{aligned}$$

The Cost of QuickSort

However, on average, there will be a fraction of the elements that are less than the pivot (and will be to its left) and a fraction of elements that are greater than the pivot (and will be to its right). It is for this reason that QUICKSORT belongs to the family of divide-and-conquer algorithms, and indeed it has a good average-case complexity.

The Cost of QuickSort

To analyze the performance of QUICKSORT it only matters the relative order of the elements to be sorted, hence we can safely assume that the input is a permutation of the elements 1 to n. Furthermore, we can concentrate in the number of comparisons between the elements since the total cost will be proportional to that number of comparisons.

The Cost of QuickSort

Let us assume that all $n!$ possible input permutations is equally likely and let q_n be the expected number of comparisons to sort the n elements. Then

$$
\begin{aligned}
q_n &= \sum_{1\le j\le n} \mathbb{E}[\#\text{ compar.}\,|\,\text{pivot is the } j\text{-th}] \times \Pr\{\text{pivot is the } j\text{-th}\} \\
&= \sum_{1\le j\le n} (n-1+q_{j-1}+q_{n-j}) \times \frac{1}{n} \\
&= n + \mathcal{O}(1) + \frac{1}{n}\sum_{1\le j\le n} (q_{j-1}+q_{n-j}) \\
&= n + \mathcal{O}(1) + \frac{2}{n}\sum_{0\le j<n} q_j
\end{aligned}
$$

To solve this recurrence we can use the theorems that we have studied before; we will use a generalization known as the *continuous master theorem* (CMT).

The Continuous Master Theorem

CMT considers divide-and-conquer recurrences of the following type:

$$F_n = t_n + \sum_{0 \le j < n} \omega_{n,j} F_j, \qquad n \ge n_0$$

for some positive integer n_0, a function t_n, called the *toll function*, and a sequence of *weights* $\omega_{n,j} \ge 0$. The weights must satisfy two conditions:

1. $W_n = \sum_{0 \le j < n} \omega_{n,j} \ge 1$ (at least one recursive call).
2. $Z_n = \sum_{0 \le j < n} \frac{j}{n} \cdot \frac{\omega_{n,j}}{W_n} < 1$ (the size of the subinstances is a fraction of the size of the original instance).

The next step is to find a *shape function* $\omega(z)$, a continuous function approximating the discrete weights $\omega_{n,j}$.

The Continuous Master Theorem

Definition

Given the sequence of weights $\omega_{n,j}$, $\omega(z)$ is a shape function for that set of weights if

1. $\int_0^1 \omega(z)\,dz \geq 1$
2. there exists a constant $\rho > 0$ such that

$$\sum_{0 \leq j < n} \left| \omega_{n,j} - \int_{j/n}^{(j+1)/n} \omega(z)\,dz \right| = \mathcal{O}(n^{-\rho})$$

A simple trick that works very often, to obtain a convenient shape function is to substitute j by $z \cdot n$ in $\omega_{n,j}$, multiply by n and take the limit for $n \to \infty$.

$$\omega(z) = \lim_{n \to \infty} n \cdot \omega_{n,z \cdot n}$$

The Continuous Master Theorem

The extension of discrete functions to functions in the real domain is immediate, e.g., $j^2 \to z^2$. For binomial numbers one might use the approximation

$$\binom{z \cdot n}{k} \sim \frac{(z \cdot n)^k}{k!}.$$

The continuation of factorials to the real numbers is given by Euler's Gamma function $\Gamma(z)$ and that of harmonic numbers by Ψ function: $\Psi(z) = \frac{d \ln \Gamma(z)}{dz}$.
For instance, in quicksort's recurrence all wright are equal: $\omega_{n,j} = \frac{2}{n}$. Hence a simple valid shape function is $\omega(z) = \lim_{n\to\infty} n \cdot \omega_{n,z\cdot n} = 2$.

The Continuous Master Theorem

Theorem (Roura, 1997)

Let F_n satisfy the recurrence

$$F_n = t_n + \sum_{0 \le j < n} \omega_{n,j} F_j,$$

with $t_n = \Theta(n^a (\log n)^b)$, for some constants $a \ge 0$ and $b > -1$, and let $\omega(z)$ be a shape function for the weights $\omega_{n,j}$. Let $\mathcal{H} = 1 - \int_0^1 \omega(z) z^a \, dz$ and $\mathcal{H}' = -(b+1) \int_0^1 \omega(z) z^a \ln z \, dz$. Then

$$F_n = \begin{cases} \frac{t_n}{\mathcal{H}} + o(t_n) & \text{if } \mathcal{H} > 0, \\ \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n) & \text{if } \mathcal{H} = 0 \text{ and } \mathcal{H}' \neq 0, \\ \Theta(n^\alpha) & \text{if } \mathcal{H} < 0, \end{cases}$$

where $x = \alpha$ is the unique non-negative solution of the equation

$$1 - \int_0^1 \omega(z) z^x \, dz = 0.$$

Solving QuickSort's Recurrence

We apply CMT to quicksort's recurrence with the set of weights $\omega_{n,j} = 2/n$ and toll function $t_n = n - 1$. As we have already seen, we can take $\omega(z) = 2$, and the CMT applies with $a = 1$ and $b = 0$. All necessary conditions to apply CMT are met. Then we compute

$$\mathcal{H} = 1 - \int_0^1 2z\,dz = 1 - \left. z^2 \right|_{z=0}^{z=1} = 0,$$

hence we will have to apply CMT's second case and compute

$$\mathcal{H}' = -\int_0^1 2z \ln z\,dz = \left. \frac{z^2}{2} - z^2 \ln z \right|_{z=0}^{z=1} = \frac{1}{2}.$$

Finally,

$$\begin{aligned} q_n &= \frac{n \ln n}{1/2} + o(n \log n) = 2n \ln n + o(n \log n) \\ &= 1.386\ldots n \log_2 n + o(n \log n). \end{aligned}$$

Part II

Divide and Conquer Algorithms

Part II

Divide and Conquer Algorithms

QuickSelect

The selection problem is to find the j-th smallest element in a given set of n elements. More specifically, given an array $A[1..n]$ of size $n > 0$ and a rank j, $1 \leq j \leq n$, the selection problem is to find the j-th element of A if it were in ascending order.
For $j = 1$ we want to find the minimum, for $j = n$ we want to find the maximum, for $j = \lceil n/2 \rceil$ we are looking for the median, etc.

QuickSelect

The problem can be trivially but inefficiently (because it implies doing much more work than needed!) solved with cost $\Theta(n \log n)$ sorting the array. Another solution keeps an unsorted table of the j smallest elements seen so far while scanning the array from left to right; it has cost $\Theta(j \cdot n)$, and using clever data structures the cost can be improved to $\Theta(n \log j)$. This is not a real improvement with respect the first trivial solution if $j = \Theta(n)$.
QUICKSELECT (Hoare, 1962), also known as FIND and as *one-sided* QUICKSORT, is a variant of QUICKSORT adapted to select the j-th smallest element out of n.

QuickSelect

Assume we partition the subarray $A[\ell..u]$, that contains the elements of ranks ℓ to u, $\ell \leq j \leq u$, with respect some pivot p. Once the partition finishes, suppose that the pivot ends at position k.
Then $A[\ell..k-1]$ contains the elements of ranks ℓ to $(k-1)$ in A and $A[k+1..u]$ contains the elements of ranks $(k+1)$ to u. If $j = k$ we are done since we have found the sought element. If $j < k$ then we need to recursively continue in the left subarray $A[\ell..k-1]$, whereas if $j > k$ then the sought element must be located in the right subarray $A[k+1..u]$.

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

9	5	10	12	3	1	11	15	7	2	8	13	6	4	14

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

9	5	10	12	3	1	11	15	7	2	8	13	6	4	14

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

7	5	4	6	3	1	8	2	9	15	11	13	12	10	14

pivot ends at position $k = 9 > j$

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

7	5	4	6	3	1	8	2	9	15	11	13	12	10	14

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

1	5	4	2	3	6	8	7	9	15	11	13	12	10	14

pivot ends at position $k = 6 > j$

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

1	5	4	2	3	6	8	7	9	15	11	13	12	10	14

QuickSelect

Example

We are looking the fourth element ($j = 4$) out of $n = 15$ elements

2	3	1	4	5	6	8	7	9	15	11	13	12	10	14

pivot ends at position $k = 4 = j \Rightarrow$ DONE!

QuickSelect

```
  procedure QUICKSELECT($A$, $\ell$, $j$, $u$)
Ensure: Returns the $(j + 1 - \ell)$-th smallest element in $A[\ell..u]$,
  $\ell \leq j \leq u$
      if $\ell = u$ then
          return $A[\ell]$
      end if
      PARTITION($A, \ell, u, k$)
      if $j = k$ then
          return $A[k]$
      end if
      if $j < k$ then
          return QUICKSELECT($A, \ell, j, k - 1$)
      else
          return QUICKSELECT($A, k + 1, j, u$)
      end if
  end procedure
```

QuickSelect

Our implementation of QUICKSELECT uses tail recursion, it is hence straightforward to derive an efficient iterative solution that needs no auxiliary memory space.

In the worst-case, the cost of QUICKSELECT is $\Theta(n^2)$. However, its average cost is $\Theta(n)$, with the proportionality constant depending on the ratio j/n. Knuth (1971) proved that $C_n^{(j)}$, the expected number of comparisons to find the smallest j-th element among n is:

$$\begin{aligned} C_n^{(j)} &= 2\big((n+1)H_n - (n+3-j)H_{n+1-j} \\ &\quad - (j+2)H_j + n + 3\big) \end{aligned}$$

The maximum average cost corresponds to finding the median ($j = \lfloor n/2 \rfloor$); then we have

$$C_n^{(\lfloor n/2 \rfloor)} = 2(\ln 2 + 1)n + o(n).$$

QuickSelect

Let us now consider the analysis of the expected cost C_n when j takes any value between 1 and n with identical probability. Then

$$\begin{aligned} &C_n = n + \mathcal{O}(1) \\ &+ \frac{1}{n} \sum_{1 \le k \le n} \mathbb{E}[\text{remaining number of comp.} \mid \text{pivot is the } k\text{-th element}]\,, \end{aligned}$$

as the pivot will be the k-th smallest element with probability $1/n$ for all k, $1 \le k \le n$.

QuickSelect

The probability that $j = k$ is $1/n$, then no more comparisons are need since we would be done. The probability that $j < k$ is $(k-1)/n$, then we will have to make C_{k-1} comparisons. Similarly, with probability $(n-k)/n$ we have $j > k$ and we will then make C_{n-k} comparisons. Thus

$$\begin{aligned} C_n &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \le k \le n} \frac{k-1}{n} C_{k-1} + \frac{n-k}{n} C_{n-k} \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \le k < n} \frac{k}{n} C_k . \end{aligned}$$

Applying the CMT with the shape function

$$\lim_{n \to \infty} n \cdot \frac{2}{n} \frac{z \cdot n}{n} = 2z$$

we obtain $\mathcal{H} = 1 - \int_0^1 2z^2 \, dz = 1/3 > 0$ and $C_n = 3n + o(n)$.

Part II

Divide and Conquer Algorithms

Rivest-Floyd's Worst-Case Linear Time Selection

We can design a selection algorithm with linear worst-case cost; we need only to guarantee that the pivot that we choose in each recursive step will divide the array into two subarrays such that their respective sizes are a fraction of the original size n. And we must pick such a pivot with cost $\mathcal{O}(n)$. Then, in the worst-case, the cost of the algorithm will be

$$C(n) = \mathcal{O}(n) + C(p \cdot n),$$

for some $p < 1$. Since $\log_{1/p} 1 = 0 < 1$ we conclude that $C(n) = \mathcal{O}(n)$. On the other hand, it is obvious that $C(n) = \Omega(n)$, hence $C(n) = \Theta(n)$.
This is the idea behind the selection algorithm proposed by Rivest and Floyd (1970).

Rivest-Floyd's Worst-Case Linear Time Selection

The only difference between Hoare's QUICKSELECT algorithm and Rivest and Floyd's algorithm is the way to choose the pivot of each recursive step.

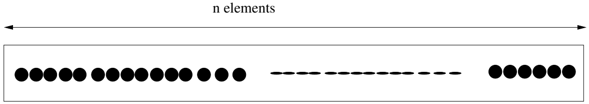
Rivest and Floyd's algorithm picks a "good" pivot using the selection algorithm recursively!

Rivest-Floyd's Worst-Case Linear Time Selection

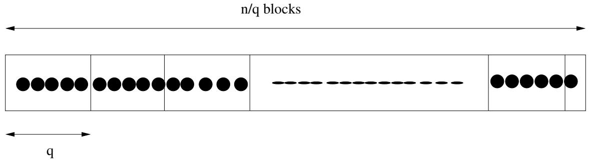
In particular, the algorithm will pick the so-called pseudo-median as the pivot.

1. The array $A[\ell..u]$ of size $n = u - \ell + 1$ is subdivided into blocks of q elements (except possibly the last block which might contain $< q$ elements) for some odd constant q. For each block we obtain the median of the q elements.
2. The selection algorithm is recursively applied to find the median of the $\lceil n/q \rceil$ medians that were obtained in the previous step.
3. The median of the medians (*pseudo-median*) is used as the pivot to partition the original subarray, and a recursive call is made in the appropriate subarray, either left or right to the pivot.

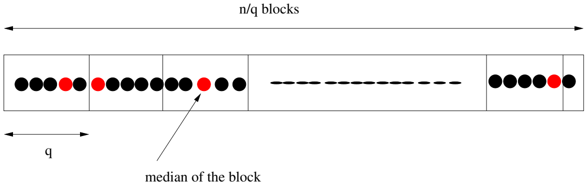
Rivest-Floyd's Worst-Case Linear Time Selection



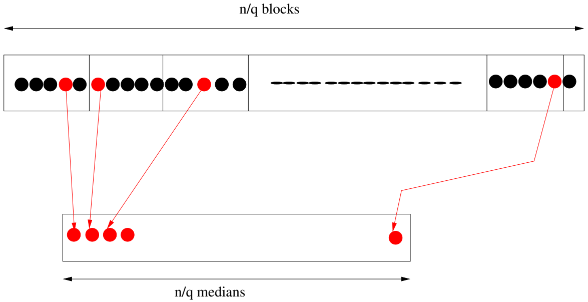
Rivest-Floyd's Worst-Case Linear Time Selection



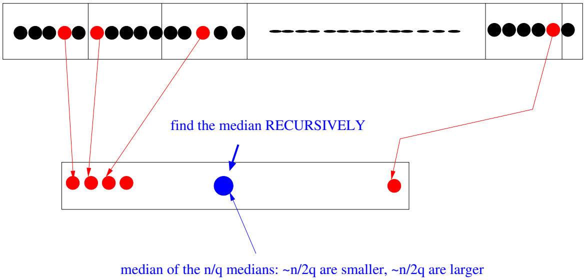
Rivest-Floyd's Worst-Case Linear Time Selection



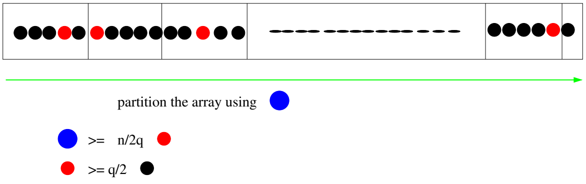
Rivest-Floyd's Worst-Case Linear Time Selection



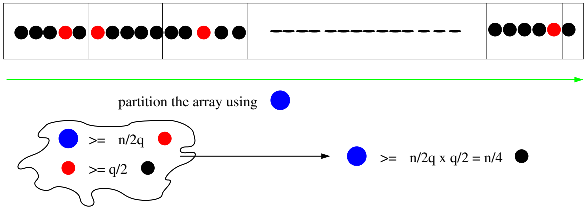
Rivest-Floyd’s Worst-Case Linear Time Selection


find the median RECURSIVELY
median of the n/q medians: ~n/2q are smaller, ~n/2q are larger

Rivest-Floyd's Worst-Case Linear Time Selection



Rivest-Floyd's Worst-Case Linear Time Selection



Rivest-Floyd's Worst-Case Linear Time Selection

The cost of the first step (computing the medians of the blocks) is $\Theta(n)$ since the cost of finding the median of each block is $\Theta(1)$ and there are $\approx n/q$ blocks. The recursive call to find the median of the medians has cost $C(n/q)$ since the input consists of the $\lceil n/q \rceil$ medians of the blocks.
Since the pivot that we pick is the median of $\lceil n/q \rceil$ medians (this is why we call it the pseudo-median of the n elements), we have the guarantee that at least $n/2q \cdot q/2 = n/4$ elements are smaller than the pivot, and similarly, at least $n/4$ elements are larger. Thus, in the worst-case, after the partition with respect to the pivot (which costs $\Theta(n)$) we will have to proceed recursively into a subarray with at most $3n/4$ elements.

Rivest-Floyd's Worst-Case Linear Time Selection

Putting everything together, the worst-case cost of the algorithm satisfies

$$C(n) = \mathcal{O}(n) + C(n/q) + C(3n/4),$$

where the $\mathcal{O}(n)$ term collects the cost of finding the medians of the blocks and the cost of partitioning the original array around the pivot. The solution to the recurrence above (one can use the *Discrete Master Theorem*) is $C(n) = \mathcal{O}(n)$ if

$$\frac{3}{4} + \frac{1}{q} < 1.$$

For instance $q = 3$ does not work, but $q = 5$, which was the value originally proposed by Rivest and Floyd, does.