On Partial Sorting

Conrado Martínez

Univ. Politècnica de Catalunya, Spain

10th Seminar on the Analysis of Algorithms
MSRI, Berkeley, U.S.A.
June 2004
1 Introduction

2 Partial Quicksort

3 Generalized Partial Sorting: Chunksort
**Partial sorting**: Given an array $A$ of $n$ elements and a value $1 \leq m \leq n$, rearrange $A$ so that its first $m$ positions contain the $m$ smallest elements in ascending order.

For $m = \Theta(n)$ it might be OK to sort the array; otherwise, we are doing too much work.
Partial sorting: Given an array $A$ of $n$ elements and a value $1 \leq m \leq n$, rearrange $A$ so that its first $m$ positions contain the $m$ smallest elements in ascending order.

For $m = \Theta(n)$ it might be OK to sort the array; otherwise, we are doing too much work.
A Few Common Solutions

- Idea #1: Partial heapsort
  - Build a heap with the $n$ elements and perform $m$ extractions of the heap’s minimum
  - The worst-case cost is $\Theta(n + m \log n)$
  - This the “traditional” implementation of C++ STL’s `partial_sort`
A Few Common Solutions

Idea #1: Partial heapsort
- Build a heap with the $n$ elements and perform $m$ extractions of the heap’s minimum
- The worst-case cost is $\Theta(n + m \log n)$
- This the “traditional” implementation of C++ STL’s `partial_sort`
A Few Common Solutions

- **Idea #1: Partial heapsort**
  - Build a heap with the $n$ elements and perform $m$ extractions of the heap’s minimum
  - The worst-case cost is $\Theta(n + m \log n)$
  - This the “traditional” implementation of C++ STL’s `partial_sort`
A Few Common Solutions

- Idea #2: On-line selection
  - Build a heap with the $m$ first elements; then scan the remaining $n - m$ elements and update the heap as needed; finally extract the $m$ elements from the heap
  - The worst-case cost is $\Theta(n \log m)$
  - Not very attractive unless $m$ is very small or if used in on-line settings
A Few Common Solutions

• **Idea #2: On-line selection**
  - Build a heap with the $m$ first elements; then scan the remaining $n - m$ elements and update the heap as needed; finally extract the $m$ elements from the heap
  - The worst-case cost is $\Theta(n \log m)$
  - Not very attractive unless $m$ is very small or if used in on-line settings
A Few Common Solutions

- **Idea #2: On-line selection**
  - Build a heap with the $m$ first elements; then scan the remaining $n - m$ elements and update the heap as needed; finally extract the $m$ elements from the heap.
  - The worst-case cost is $\Theta(n \log m)$.
  - Not very attractive unless $m$ is very small or if used in on-line settings.
**A Few Common Solutions**

- **Idea #3: Quickselsort**
  - Find the $m$th smallest element with quickselect, then quicksort the preceding $m - 1$ elements
  - The average cost is $\Theta(n + m \log m)$
  - Uses two basic algorithms widely available (and highly tuned for performance in standard libraries)
A Few Common Solutions

- **Idea #3: Quickselsort**
  - Find the $m$th smallest element with quickselect, then quicksort the preceding $m - 1$ elements
  - The average cost is $\Theta(n + m \log m)$
  - Uses two basic algorithms widely available (and highly tuned for performance in standard libraries)
A Few Common Solutions

- **Idea #3: Quickselsort**
  - Find the $m$th smallest element with quickselect, then quicksort the preceding $m-1$ elements
  - The average cost is $\Theta(n + m \log m)$
  - Uses two basic algorithms widely available (and highly tuned for performance in standard libraries)
Introduction

2 Partial Quicksort

Generalized Partial Sorting: Chunksort
void partial_quicksort(vector<Elem>& A, int i, int j, int m) {
    if (i < j) {
        int p = get_pivot(A, i, j);
        swap(A[p], A[1]);
        int k;
        partition(A, i, j, k);
        partial_quicksort(A, i, k - 1, m);
        if (k < m - 1)
            partial_quicksort(A, k + 1, j, m);
    }
}
Probability that the selected pivot is the $k$-th of $n$ elements:

$\pi_{n,k}$

Average number of comparisons $P_{n,m}$ to sort the $m$ smallest elements out of $n$:

$$P_{n,m} = n - 1 + \sum_{k=m+1}^{n} \pi_{n,k} \cdot P_{k-1,m}$$

$$+ \sum_{k=1}^{m} \pi_{n,k} \cdot (P_{k-1,k-1} + P_{n-k,m-k})$$
The Analysis

- Probability that the selected pivot is the $k$-th of $n$ elements:
  $\pi_{n,k}$

- Average number of comparisons $P_{n,m}$ to sort the $m$ smallest elements out of $n$:

$$P_{n,m} = n - 1 + \sum_{k=m+1}^{n} \pi_{n,k} \cdot P_{k-1,m}$$

$$+ \sum_{k=1}^{m} \pi_{n,k} \cdot (P_{k-1,k-1} + P_{n-k,m-k})$$
For \( m = n \), partial quicksort \( \equiv \) quicksort; let \( q_n \) denote the average number of comparisons used by quicksort.

Hence,

\[
P_{n,m} = n - 1 + \sum_{0 \leq k < m} \pi_{n,k+1} \cdot q_k + \sum_{k=m+1}^{n} \pi_{n,k} \cdot P_{k-1,m} + \sum_{k=1}^{m} \pi_{n,k} \cdot P_{n-k,m-k} \tag{1}
\]
For $m = n$, partial quicksort $\equiv$ quicksort; let $q_n$ denote the average number of comparisons used by quicksort.

Hence,

$$P_{n,m} = n - 1 + \sum_{0 \leq k < m} \pi_{n,k+1} \cdot q_k$$

$$+ \sum_{k=m+1}^{n} \pi_{n,k} \cdot P_{k-1,m} + \sum_{k=1}^{m} \pi_{n,k} \cdot P_{n-k,m-k} \quad (1)$$
The Analysis

- The recurrence for $P_{n,m}$ is the same as for quickselect but the toll function is

$$t_{n,m} = n - 1 + \sum_{0 \leq k < m} \pi_{n,k+1} \cdot q_k$$

- Up to now, everything holds no matter which pivot selection scheme do we use; for the standard variant we must take $\pi_{n,k} = 1/n$, for all $1 \leq k \leq n$
The Analysis

- The recurrence for $P_{n,m}$ is the same as for quickselect but the toll function is

$$t_{n,m} = n - 1 + \sum_{0 \leq k < m} \pi_{n,k+1} \cdot q_k$$

- Up to now, everything holds no matter which pivot selection scheme do we use; for the standard variant we must take $\pi_{n,k} = 1/n$, for all $1 \leq k \leq n$
Define the two BGFs

\[ P(z, u) = \sum_{n \geq 0} \sum_{1 \leq m \leq n} P_{n,m} z^n u^m \]
\[ T(z, u) = \sum_{n \geq 0} \sum_{1 \leq m \leq n} t_{n,m} z^n u^m \]

Then the recurrence (1) translates to

\[ \frac{\partial P}{\partial z} = \frac{P(z, u)}{1 - z} + \frac{u P(z, u)}{1 - uz} + \frac{\partial T}{\partial z} \]  \hspace{1cm} (2)
The Analysis: Generating Functions

- Define the two BGFs

\[ P(z, u) = \sum_{n \geq 0} \sum_{1 \leq m \leq n} P_{n,m} z^n u^m \]
\[ T(z, u) = \sum_{n \geq 0} \sum_{1 \leq m \leq n} t_{n,m} z^n u^m \]

- Then the recurrence (1) translates to

\[ \frac{\partial P}{\partial z} = \frac{P(z, u)}{1 - z} + \frac{u P(z, u)}{1 - uz} + \frac{\partial T}{\partial z} \]
The Analysis: Generating Functions

- Let \( P(z, u) = F(z, u) + S(z, u) \), where \( F(z, u) \) corresponds to the selection part of the toll function \((n - 1)\) and \( S(z, u) \) to the sorting part \( \sum_k q_k / n \)

- Let

\[
T_F(z, u) = \sum_{n \geq 0} \sum_{1 \leq m \leq n} (n - 1)z^n u^m
\]

\[
T_S(z, u) = \sum_{n \geq 0} \sum_{1 \leq m \leq n} \frac{1}{n} \left( \sum_{0 \leq k < m} q_k \right) z^n u^m
\]
Let \( P(z, u) = F(z, u) + S(z, u) \), where \( F(z, u) \) corresponds to the selection part of the toll function \((n - 1)\) and \( S(z, u) \) to the sorting part \((\sum_k q_k/n)\).

Let

\[
T_F(z, u) = \sum_{n \geq 0} \sum_{1 \leq m \leq n} (n - 1)z^nu^m
\]

\[
T_S(z, u) = \sum_{n \geq 0} \sum_{1 \leq m \leq n} \frac{1}{n} \left( \sum_{0 \leq k < m} q_k \right) z^nu^m
\]
Then, each of $F(z, u)$ and $S(z, u)$ satisfies a differential equation like (2) and

$$F(z, u) = \frac{1}{(1 - z)(1 - zu)} \times \left\{ \int (1 - z)(1 - zu) \frac{\partial T_F}{\partial z} \ dz + K_F \right\}$$

$$S(z, u) = \frac{1}{(1 - z)(1 - zu)} \times \left\{ \int (1 - z)(1 - zu) \frac{\partial T_S}{\partial z} \ dz + K_S \right\}$$
The Analysis: Generating Functions

- $F(z, u)$ satisfies exactly the same differential equation as standard quickselect; it is well known (Knuth, 1971) that for $1 \leq m \leq n$,

$$F_{n,m} = [z^n u^m] F(z, u) = 2 \left( n + 3 + (n + 1)H_n - (m + 2)H_m - (n + 3 - m)H_{n+1-m} \right)$$
To compute $S(z, u)$, we need first to determine $T_S(z, u)$

$$
\frac{\partial T_S}{\partial z} = \frac{u}{1 - z} \frac{Q(uz)}{1 - uz}
$$

where $Q(z) = \sum_{n \geq 0} q_n z^n$.

With the toll function $n - 1$, we solve the recurrence for quicksort to get

$$
Q(z) = \frac{2}{(1 - z)^2} \left( \ln \frac{1}{1 - z} - z \right)
$$
To compute $S(z, u)$, we need first to determine $T_S(z, u)$

$$\frac{\partial T_S}{\partial z} = \frac{u}{1-z} \frac{Q(uz)}{1 - uz}$$

where $Q(z) = \sum_{n \geq 0} q_n z^n$.

With the toll function $n - 1$, we solve the recurrence for quicksort to get

$$Q(z) = \frac{2}{(1-z)^2} \left( \ln \frac{1}{1-z} - z \right)$$
Hence,

\[ S(z, u) = \frac{1}{(1 - z)(1 - uz)} \left\{ \int u Q(uz) \, dz + K_S \right\} \]

\[ = \frac{2}{(1 - uz)^2(1 - z)} \ln \frac{1}{1 - uz} \]

\[ + \frac{2}{(1 - z)(1 - uz)} \ln \frac{1}{1 - uz} \]

\[ - 4 \frac{uz}{(1 - uz)^2(1 - z)} \]
The Analysis: Generating Functions

- Extracting coefficients $S_{n,m} = [z^n u^m] S(z, u)$

$$S_{n,m} = 2(m + 1)H_m - 6m + 2H_m$$

- And finally

$$P_{n,m} = 2n + 2(n + 1)H_n - 2(n + 3 - m)H_{n+1-m} - 6m + 6$$
The Analysis: Generating Functions

- Extracting coefficients $S_{n,m} = [z^n u^m] S(z, u)$

$$S_{n,m} = 2(m + 1)H_m - 6m + 2H_m$$

- And finally

$$P_{n,m} = 2n + 2(n + 1)H_n - 2(n + 3 - m)H_{n+1-m} - 6m + 6$$
The average number of comparisons made by quickselsort is

\[ Q_{n,m} = F_{n,m} + q_{m-1} \]

Using partial quicksort we save

\[ Q_{n,m} - P_{n,m} = 2m - 4H_m + 2 \]

comparisons on the average.
Partial quicksort vs. quickselsort

- The average number of comparisons made by quickselsort is
  \[ Q_{n,m} = F_{n,m} + q_{m-1} \]

- Using partial quicksort we save
  \[ Q_{n,m} - P_{n,m} = 2m - 4H_m + 2 \]

comparisons on the average
To analyze other quantities, e.g., the average number of exchanges, we set up solve recurrence (1) with the toll function

\[ t_{n,m} = a \cdot n + b + \frac{1}{n} \sum_{0 \leq k < m} q'_k \]

and with \( q'_n \) the solution of

\[ q'_n = a \cdot n + b + \frac{2}{n} \sum_{0 \leq k < n} q'_k \]
If we compare partial quicksort with quickselsort w.r.t. to the generalized toll function we obtain that difference is

\[ 2am + (b - 3a)H_m + a - b \]

If we consider exchanges then \( a = 1/6 \) and \( b = -1/3 \); partial quicksort saves on average

\[ \frac{m}{3} - \frac{5}{6}H_m + \frac{1}{2} \]
Partial quicksort vs. quickselsort

If we compare partial quicksort with quickselsort w.r.t. to the generalized toll function we obtain that difference is

$$2am + (b - 3a)H_m + a - b$$

If we consider exchanges then $a = 1/6$ and $b = -1/3$; partial quicksort saves on average

$$\frac{m}{3} - \frac{5}{6}H_m + \frac{1}{2}$$
Partial quicksort avoids some of the redundant comparisons, exchanges, ... made by quickselsort

- It is easily implemented
- It benefits from standard optimization techniques: sampling, recursion removal, recursion cutoff on small subfiles, improved partitioning schemas, etc.
- The same idea can be applied to similar algorithms like radix sorting and quicksort for strings
Final remarks on partial quicksort

- Partial quicksort avoids some of the redundant comparisons, exchanges, ... made by quickselsort
- It is easily implemented
  - It benefits from standard optimization techniques: sampling, recursion removal, recursion cutoff on small subfiles, improved partitioning schemes, etc.
  - The same idea can be applied to similar algorithms like radix sorting and quicksort for strings
Final remarks on partial quicksort

- Partial quicksort avoids some of the redundant comparisons, exchanges, ... made by quickselsort
- It is easily implemented
- It benefits from standard optimization techniques: sampling, recursion removal, recursion cutoff on small subfiles, improved partitioning schemes, etc.
- The same idea can be applied to similar algorithms like radix sorting and quicksort for strings
Final remarks on partial quicksort

- Partial quicksort avoids some of the redundant comparisons, exchanges, ... made by quickselect.
- It is easily implemented.
- It benefits from standard optimization techniques: sampling, recursion removal, recursion cutoff on small subfiles, improved partitioning schemes, etc.
- The same idea can be applied to similar algorithms like radix sorting and quicksort for strings.
Introduction

Partial Quicksort

3 Generalized Partial Sorting: Chunksort
Given $J_1 = [\ell_1, u_1], J_2 = [\ell_2, u_2], \ldots, J_p = [\ell_p, u_p]$ the goal is to rearrange the array $A[1..n]$ so that

$$A[1..\ell_1 - 1] \leq A[\ell_1..u_1] \leq A[u_1 + 1..\ell_2 - 1] \leq \cdots \leq A[\ell_p..u_p] \leq A[u_p + 1..n]$$

and each $A[\ell_j..u_j], 1 \leq j \leq p$, is sorted in ascending order.

The same principles can be used to rearrange and “cluster” the items in $A$ given $p$ key intervals $[K_1, K'_1], [K_2, K'_2], \ldots, [K_p, K'_p]$.
Given \( J_1 = [\ell_1, u_1], J_2 = [\ell_2, u_2], \ldots, J_p = [\ell_p, u_p] \) the goal is to rearrange the array \( A[1..n] \) so that

\[
A[1..\ell_1 - 1] \leq A[\ell_1..u_1] \leq A[u_1 + 1..\ell_2 - 1] \leq \cdots \leq A[\ell_p..u_p] \leq A[u_p + 1..n]
\]

and each \( A[\ell_j..u_j], 1 \leq j \leq p, \) is sorted in ascending order.

The same principles can be used to rearrange and “cluster” the items in \( A \) given \( p \) key intervals \([K_1, K_1'], [K_2, K_2'], \ldots, [K_p, K_p']\).
void chunksort(vector<T>& A, vector<int>& I, 
    int i, int j, int l, int u) {
    if (i >= j) return;
    if (l <= u) {
        int k;
        partition(A, i, j, k);
        int r = locate(I, l, u, k);
        // locate the value r such that \( I[r] \leq k < I[r+1] \)
        if (r % 2 == 0) { // \( r = 2t \implies I[r] = u_t \leq k < \ell_{t+1} \)
            chunksort(A, I, i, k - 1, l, r);
            chunksort(A, I, k + 1, j, r + 1, u);
        } else { // \( r = 2t - 1 \implies I[r] = \ell_t \leq k < u_t \)
            // this can be optimized
            chunksort(A, I, i, k - 1, l, r);
            chunksort(A, I, k + 1, j, r, u);
        }
    }
}
With $p = 1$, $\ell_1 = 1$ and $u_1 = n$, chunksort sorts the array; it is equivalent to quicksort.

Setting $p = 1$ and $\ell_1 = u_1 = m$; chunksort selects the $m$th smallest element in $A$.

If $p = 1$, $\ell_1 = 1$ and $u_1 = m \leq n$, chunksort partially sorts the array.

We can also select multiple ranks by setting $\ell_j = u_j$ for $1 \leq j \leq p$; chunksort behaves like multiple quickselect then...
Chunksort

- With $p = 1$, $\ell_1 = 1$ and $u_1 = n$, chunksort sorts the array; it is equivalent to quicksort.
- Setting $p = 1$ and $\ell_1 = u_1 = m$; chunksort selects the $m$th smallest element in $A$.
- If $p = 1$, $\ell_1 = 1$ and $u_1 = m \leq n$, chunksort partially sorts the array.
- We can also select multiple ranks by setting $\ell_j = u_j$ for $1 \leq j \leq p$; chunksort behaves like multiple quickselect then...
Chunksort

- With $p = 1$, $\ell_1 = 1$ and $u_1 = n$, chunksort sorts the array; it is equivalent to quicksort.
- Setting $p = 1$ and $\ell_1 = u_1 = m$; chunksort selects the $m$th smallest element in $A$.
- If $p = 1$, $\ell_1 = 1$ and $u_1 = m \leq n$, chunksort partially sorts the array.
- We can also select multiple ranks by setting $\ell_j = u_j$ for $1 \leq j \leq p$; chunksort behaves like multiple quickselect then.
With $p = 1$, $\ell_1 = 1$ and $u_1 = n$, chunksort sorts the array; it is equivalent to quicksort.

Setting $p = 1$ and $\ell_1 = u_1 = m$; chunksort selects the $m$th smallest element in $A$.

If $p = 1$, $\ell_1 = 1$ and $u_1 = m \leq n$, chunksort partially sorts the array.

We can also select multiple ranks by setting $\ell_j = u_j$ for $1 \leq j \leq p$; chunksort behaves like multiple quickselect then...
Let $m_k = u_k - l_k + 1$ denote the size of the $k$th interval, $\bar{m}_k = l_{k+1} - u_k - 1$ the size of the $k$th gap, and $m = m_1 + \cdots + m_p$.

Let $C_n$ denote the average number of key comparisons needed by chunksort to sort the keys in the intervals $J_1, J_2, \ldots, J_p$. Then

$$C_n = 2n + u_p - l_1 + 2(n + 1)H_n - 7m - 2 + 15p - 2(l_1 + 2)H_{l_1} - 2(n + 3 - u_p)H_{n+1-u_p} - 2 \sum_{k=1}^{p-1}(\bar{m}_k + 5)H_{\bar{m}_k}.$$
Let $m_k = u_k - \ell_k + 1$ denote the size of the $k$th interval, $\overline{m}_k = \ell_{k+1} - u_k - 1$ the size of the $k$th gap, and $m = m_1 + \cdots + m_p$.

Let $C_n$ denote the average number of key comparisons needed by chunksort to sort the keys in the intervals $J_1, J_2, \ldots, J_p$. Then

$$C_n = 2n + u_p - \ell_1 + 2(n + 1)H_n - 7m - 2 + 15p$$

$$- 2(\ell_1 + 2)H_{\ell_1} - 2(n + 3 - u_p)H_{n+1-u_p} - 2 \sum_{k=1}^{p-1} (\overline{m}_k + 5)H_{\overline{m}_k}$$
“Filtering out outliers”: $p = 1$, $\ell_1 = \alpha n$, $u_1 = \beta n$, with $0 < \alpha < \beta \leq 1 - \alpha < 1$

Let $Q_n(\alpha, \beta)$ the number of comparisons needed to solve the problem using quickselect (twice) plus quicksort.

Then

$$Q_n(\alpha, \beta) - C_n = 2(1 - 2\alpha + \beta)n + o(n)$$
“Filtering out outliers”: \( p = 1, \ell_1 = \alpha n, u_1 = \beta n, \) with \( 0 < \alpha < \beta \leq 1 - \alpha < 1 \)

Let \( Q_n(\alpha, \beta) \) the number of comparisons needed to solve the problem using quickselect (twice) plus quicksort

Then

\[
Q_n(\alpha, \beta) - C_n = 2(1 - 2\alpha + \beta)n + o(n)
\]
“Filtering out outliers”: \( p = 1, \ell_1 = \alpha n, u_1 = \beta n, \) with
\[0 < \alpha < \beta \leq 1 - \alpha < 1\]

Let \( Q_n(\alpha, \beta) \) the number of comparisons needed to solve the problem using quickselect (twice) plus quicksort

Then
\[Q_n(\alpha, \beta) - C_n = 2(1 - 2\alpha + \beta)n + o(n)\]
“Selecting an \( \alpha \)-cluster”: \( p = 1, \ell_1 = \alpha n - f(n), \)
\( u_1 = \alpha n + f(n) \), for some \( f(n) = o(n/\log n) \) and \( 0 < \alpha \leq 1/2 \)

Using chunksort instead of quickselect+quicksort saves

\[
2(1 - \alpha)n + 6f(n)
\]

comparisons
“Selecting an $\alpha$-cluster”: $p = 1$, $\ell_1 = \alpha n - f(n)$,
$u_1 = \alpha n + f(n)$, for some $f(n) = o(n/\log n)$ and $0 < \alpha \leq 1/2$

Using chunksort instead of quickselect+quicksort saves

$$2(1 - \alpha)n + 6f(n)$$

comparisons
Partial quicksort and chunksort are nice examples of the simplicity and elegance of the divide-and-conquer principle. Their analysis poses the same type of mathematical challenges as quicksort and quickselect do. The analysis of partial quicksort is basically identical to that of quickselect, but with a different toll function.
Partial quicksort and chunksort are nice examples of the simplicity and elegance of the divide-and-conquer principle. Their analysis poses the same type of mathematical challenges as quicksort and quickselect do. The analysis of partial quicksort is basically identical to that of quickselect, but with a different toll function.
Partial quicksort and chunksort are nice examples of the simplicity and elegance of the divide-and-conquer principle.

Their analysis poses the same type of mathematical challenges as quicksort and quickselect do.

The analysis of partial quicksort is basically identical to that of quickselect, but with a different toll function.
Likewise, chunksort can be analyzed using the same techniques as in the analysis of multiple quickselect (e.g., Prodinger, 1995).

Variants of these algorithms, like median-of-$(2t + 1)$ pivot selection, should be used in practice; but their analysis is probably difficult and cumbersome.

More real applications for chunksort?
Likewise, chunksort can be analyzed using the same techniques as in the analysis of multiple quickselect (e.g., Prodinger, 1995).

Variants of these algorithms, like median-of-$(2t + 1)$ pivot selection, should be used in practice; but their analysis is probably difficult and cumbersome.

More real applications for chunksort?
Likewise, chunksort can be analyzed using the same techniques as in the analysis of multiple quickselect (e.g., Prodinger, 1995).

Variants of these algorithms, like median-of-$(2t + 1)$ pivot selection, should be used in practice; but their analysis is probably difficult and cumbersome.

More real applications for chunksort?