Chunksort: A Generalized Partial Sort Algorithm

Conrado Martínez

Univ. Politècnica de Catalunya, Spain

Ottawa-Carleton Discrete Math Day
May 2006

Generalized partial sorting

The problem:

- The input: An array A of n elements and p intervals $I_1 = [\ell_1, u_1]$, $I_2 = [\ell_2, u_2]$, $\ldots$, $I_p = [\ell_p, u_p]$

Generalized partial sorting

The problem:

- The input: An array A of n elements and p intervals $I_1 = [\ell_1, u_1]$, $I_2 = [\ell_2, u_2]$, $\ldots$, $I_p = [\ell_p, u_p]$
- The task: To rearrange A in such a way that the blocks defined by the intervals and the gaps are in increasing order with respect to each other, and additionally, each block is also sorted.

Generalized partial sorting

The problem:

- The input: An array A of n elements and p intervals $I_1 = [\ell_1, u_1], I_2 = [\ell_2, u_2], \ldots, I_p = [\ell_p, u_p]$
- The task: To rearrange A in such a way that the blocks defined by the intervals and the gaps are in increasing order with respect to each other, and additionally, each block is also sorted.

Example

$p = 2,\ I_1 = [5, 8],\ I_2 = [12, 12]$

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

Generalized partial sorting

The problem:

- The input: An array A of n elements and p intervals $I_1 = [\ell_1, u_1], I_2 = [\ell_2, u_2], \ldots, I_p = [\ell_p, u_p]$
- The task: To rearrange A in such a way that the blocks defined by the intervals and the gaps are in increasing order with respect to each other, and additionally, each block is also sorted.

Example

$p = 2,\ I_1 = [5, 8],\ I_2 = [12, 12]$

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

⟵ gap ⟶ ⟵ block ⟶ ⟵ gap ⟶ ...

Generalized partial sorting

- Sorting the array solves the problem, but it might do much more work than actually needed, in particular, if $m = |I_1| + |I_2| + \ldots + |I_p| = o(n)$

Generalized partial sorting

- Sorting the array solves the problem, but it might do much more work than actually needed, in particular, if $m = |I_1| + |I_2| + \ldots + |I_p| = o(n)$
- This problem generalizes several well-known problems:

Generalized partial sorting

- Sorting the array solves the problem, but it might do much more work than actually needed, in particular, if $m = |I_1| + |I_2| + \ldots + |I_p| = o(n)$
- This problem generalizes several well-known problems:
    - Sorting: Take $p = 1$, $I_1 = [1, n]$

Generalized partial sorting

- Sorting the array solves the problem, but it might do much more work than actually needed, in particular, if $m = |I_1| + |I_2| + \ldots + |I_p| = o(n)$
- This problem generalizes several well-known problems:
  - Sorting: Take $p = 1$, $I_1 = [1, n]$
  - Selection: Take $p = 1$, $I_1 = [j, j]$

Generalized partial sorting

- Sorting the array solves the problem, but it might do much more work than actually needed, in particular, if $m = |I_1| + |I_2| + \ldots + |I_p| = o(n)$
- This problem generalizes several well-known problems:
  - Sorting: Take $p = 1$, $I_1 = [1, n]$
  - Selection: Take $p = 1$, $I_1 = [j, j]$
  - Multiple selection: $I_1 = [j_1, j_1], \ldots, I_p = [j_p, j_p]$

Generalized partial sorting

- Sorting the array solves the problem, but it might do much more work than actually needed, in particular, if $m = |I_1| + |I_2| + \ldots + |I_p| = o(n)$
- This problem generalizes several well-known problems:
  - Sorting: Take $p = 1$, $I_1 = [1, n]$
  - Selection: Take $p = 1$, $I_1 = [j, j]$
  - Multiple selection: $I_1 = [j_1, j_1], \ldots, I_p = [j_p, j_p]$
  - Partial sorting: $p = 1$, $I_1 = [1, m]$

Quicksort and relatives

- Quicksort and quickselect were invented in the early sixties by C.A.R. Hoare (Hoare, 1961; Hoare, 1962)

Quicksort and relatives

- Quicksort and quickselect were invented in the early sixties by C.A.R. Hoare (Hoare, 1961; Hoare, 1962)
- They are simple, elegant, beatiful and practical divide-and-conquer solutions to sorting and selection

Quicksort and relatives

- Quicksort and quickselect were invented in the early sixties by C.A.R. Hoare (Hoare, 1961; Hoare, 1962)
- They are simple, elegant, beatiful and practical divide-and-conquer solutions to sorting and selection
- Multiple quickselect uses the divide-an-conquer principle twice to solve the multiple selection problem (Prodinger, 1995)

Quicksort and relatives

- Quicksort and quickselect were invented in the early sixties by C.A.R. Hoare (Hoare, 1961; Hoare, 1962)
- They are simple, elegant, beatiful and practical divide-and-conquer solutions to sorting and selection
- Multiple quickselect uses the divide-an-conquer principle twice to solve the multiple selection problem (Prodinger, 1995)
- Partial quicksort is a slight variation of quicksort that efficiently solves the partial sorting problem (Martínez, 2004)

Quicksort

```
void quicksort(vector<Elem>& A, int i, int j) {
    if (i < j) {
       int p = select_pivot(A, i, j);
       swap(A[p], A[l]);
       int k;
       partition(A, i, j, k);
       // A[i..k-1] ≤ A[k] ≤ A[k+1..j]
       quicksort(A, i, k - 1);
       quicksort(A, k + 1, j);
} }
```

Quickselect

```
Elem quickselect(vector<Elem>& A,
                 int i, int j, int m) {
   if (i >= j) return A[i];
   int p = select_pivot(A, i, j, m);
   swap(A[p], A[l]);
   int k;
   partition(A, i, j, k);
   if (m < k)        quickselect(A, i, k - 1, m);
   else if (m > k) quickselect(A, k + 1, j, m);
   else              return A[k];
}
```

Quicksort: The recurrence for average cost

- Probability that the selected pivot is the k-th of n elements: $\pi_{n,k}$
- Average number of comparisons Q_n to sort n elements:

$$Q_n = n - 1 + \sum_{k=1}^{n} \pi_{n,k} \cdot (Q_{k-1} + Q_{n-k})$$

Quicksort: The average cost

- For the standard variant, the splitting probabilities are $\pi_{n,k} = 1/n$

Quicksort: The average cost

- For the standard variant, the splitting probabilities are $\pi_{n,k} = 1/n$
- Average number of comparisons Q_n to sort n elements (Hoare, 1962):

$$Q_n = 2(n+1)H_n - 4n$$
$$= 2n \ln n + (2\gamma - 4)n + 2 \ln n + \mathcal{O}(1)$$

where $H_n = \sum_{1 \le k \le n} 1/k = \ln n + \mathcal{O}(1)$ is the n-th harmonic number.

Quickselect: The recurrence for average cost

- Average number of comparisons $C_{n,j}$ to select the j-th out of n:

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

Quickselect: The average cost

- Average number of comparisons $C_{n,j}$ to select the j-th out of n elements (Knuth, 1971):

$$C_{n,j} = 2\big(n + 3 + (n+1)H_n - (n+3-j)H_{n+1-j} - (j+2)H_j\big).$$

Quickselect: The average cost

- Average number of comparisons $C_{n,j}$ to select the j-th out of n elements (Knuth, 1971):

$$C_{n,j} = 2(n + 3 + (n+1)H_n \\ - (n + 3 - j)H_{n+1-j} - (j+2)H_j).$$

- This is $\Theta(n)$ for any j, $1 \leq j \leq n$.

Partial quicksort

```
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[l]);
       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);
}  }
```

Partial quicksort: The average cost

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

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

Partial quicksort: The average cost

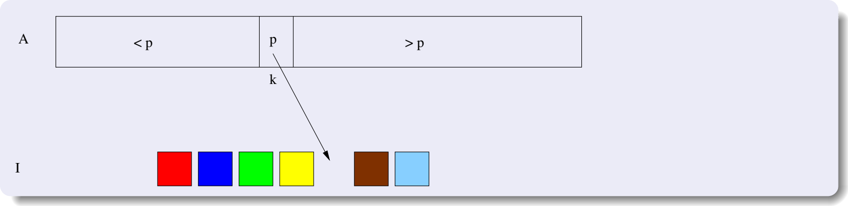
- Average number of comparisons $P_{n,m}$ to sort the m smallest elements out of n:

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

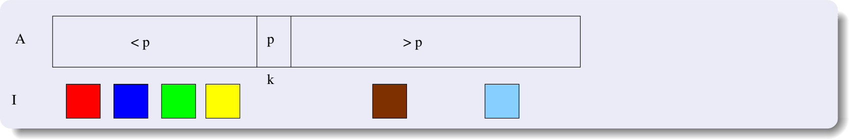
- The solution is (Martínez, 2004):

$$P_{n,m} = 2n + 2(n+1)H_n - 2(n+3-m)H_{n+1-m} - 6m + 6$$

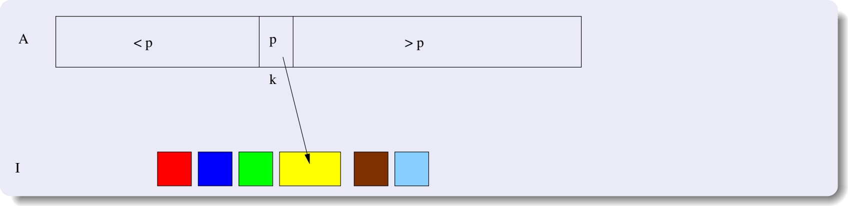
Chunksort: An example



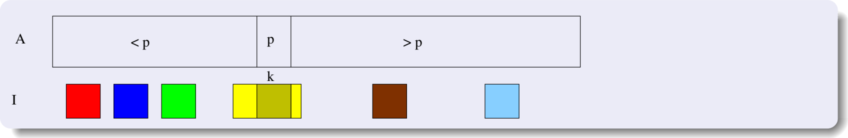
Chunksort: An example



Chunksort: An example



Chunksort: An example



Chunksort

```
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] ≤ k < I[r+1]
    if (r % 2 == 0) {
      // r = 2t  ⟹  I[r] = u_t ≤ k < ℓ_{t+1}
      chunksort(A, I, i, k - 1, l, r);
      chunksort(A, I, k + 1, j, r + 1, u);
    } else {
      // r = 2t-1  ⟹  I[r] = ℓ_t ≤ k < u_t
      chunksort(A, I, i, k - 1, l, r + 1);
      chunksort(A, I, k + 1, j, r, u);
}}}
```

Chunksort

Example (Using chunksort for partial sorting)

If $p = 1$, $I_1 = [1, m]$ then $r = 1$ whenever $k < m$; hence we make the calls

```
chunksort(A, I, i, k - 1, 1, 2);
chunksort(A, I, k + 1, j, 1, 2);
```

If $k \geq m$ then $r = 2$ and then we make the calls

```
chunksort(A, I, i, k - 1, 1, 2);
chunksort(A, I, k+1, j, 3, 2);
```

Chunksort

Example (Using chunksort for selection)

If $p = 1$, $I_1 = [j, j]$ then we will have $r = 0$ whenever $k < j$ so we call

```
chunksort(A, I, i, k - 1, 1, 0);
chunksort(A, I, k + 1, j, 1, 2);
```

If $k \geq j$ then $r = 2$ and then we make the calls

```
chunksort(A, I, i, k - 1, 1, 2);
chunksort(A, I, k + 1, j, 3, 2);
```

1. Introduction
2. The algorithm
3. The analysis
4. Conclusions

The recurrence

- We only count element comparisons

The recurrence

- We only count element comparisons
- Each partitioning stage needs $n-1$ comparisons of the pivot with all the other elements

The recurrence

- We only count element comparisons
- Each partitioning stage needs $n-1$ comparisons of the pivot with all the other elements
- We assume that pivots are chosen at random ($\pi_{n,k} = 1/n$)

The recurrence

- We only count element comparisons
- Each partitioning stage needs $n-1$ comparisons of the pivot with all the other elements
- We assume that pivots are chosen at random ($\pi_{n,k} = 1/n$)
- $C_{i,j}(\{I_r, \ldots, I_s\})$ = the average number of comparisons needed to process the subarray $A[i..j]$ for the given set of intervals $\{I_r, \ldots, I_s\}$, with $i \leq \ell_r$ and $u_s \leq j$

The recurrence

$$
\begin{aligned}
C_{i,j}(\{I_r,\ldots,I_s\}) &= n-1 \\
&+ \sum_{t=r}^{s-1}\Bigg[\sum_{\ell_t\le k<u_t}\pi_{n,k}\big(C_{i,k-1}(\{I_r,\ldots,I'_t\})+C_{k+1,j}(\{I''_t,\ldots,I_s\})\big) \\
&\qquad + \sum_{u_t\le k<\ell_{t+1}}\pi_{n,k}\big(C_{i,k-1}(\{I_r,\ldots,I_t\})+C_{k+1,j}(\{I_{t+1},\ldots,I_s\})\big)\Bigg] \\
&+ \sum_{i\le k<\ell_r}\pi_{n,k}C_{k+1,j}(\{I_r,\ldots,I_s\}) \\
&+ \sum_{\ell_s\le k\le j}\pi_{n,k}C_{i,k-1}(\{I_r,\ldots,I_s\}),
\end{aligned}
$$

with $I'_t = [\ell_t, k-1]$ and $I''_t = [k+1, u_t]$.

How to solve the recurrence ...

- We can solve this problem "iteratively", using generating functions

How to solve the recurrence ...

- We can solve this problem "iteratively", using generating functions
- First we have $p = 1$ and $I_1 = [a, b]$ and we translate the recurrence for $C_{i,j}(\{[a, b]\})$ into a functional equation for

$$C_{[a,b]}(u, v) = \sum_{1 \le i \le j} C_{i,j}(\{[a, b]\}) u^i v^j,$$

which is actually a first-order linear differential equation

How to solve the recurrence ...

- Then you can do a similar thing for $p = 2$, by introducing

$$C_{[a,b],[c,d]}(u,v) = \sum_{1 \le i \le j} C_{i,j}(\{[a,b],[c,d]\})u^i v^j,$$

which satisfies a similar ODE but the independent term now involves $C_{[a,b]}(u,v)$ and $C_{[c,d]}(u,v)$

How to solve the recurrence ...

- Then you can do a similar thing for $p = 2$, by introducing

$$C_{[a,b],[c,d]}(u,v) = \sum_{1 \le i \le j} C_{i,j}(\{[a,b],[c,d]\}) u^i v^j,$$

  which satisfies a similar ODE but the independent term now involves $C_{[a,b]}(u,v)$ and $C_{[c,d]}(u,v)$
- A pattern emerges here, so that one can obtain a general form for the functional equation satisfied by $C_{\{I_1,\ldots,I_p\}}(u,v)$

How to solve the recurrence ...

- Then you can do a similar thing for $p = 2$, by introducing

$$C_{[a,b],[c,d]}(u,v) = \sum_{1 \le i \le j} C_{i,j}(\{[a,b],[c,d]\}) u^i v^j,$$

  which satisfies a similar ODE but the independent term now involves $C_{[a,b]}(u,v)$ and $C_{[c,d]}(u,v)$
- A pattern emerges here, so that one can obtain a general form for the functional equation satisfied by $C_{\{I_1,\dots,I_p\}}(u,v)$
- Solve and extract $[u^i v^j] C_{\dots}(u,v)$

. . . But how I actually did solve it

I guessed the solution from the known solutions to the algorithms which chunksort generalizes and I proved it by induction. . .

Theorem

The average number of element comparisons $C_n(\{I_1,\ldots,I_p\}) \equiv C_{1,n}(\{I_1,\ldots,I_p\})$ needed by chunksort given the intervals $\{I_1,\ldots,I_p\}$ is

$$\begin{aligned}C_n &= 2n + u_p - \ell_1 + 2(n+1)H_n - 7m - 2 + 15p\\&\quad - 2(\ell_1+2)H_{\ell_1} - 2(n+3-u_p)H_{n+1-u_p}\\&\quad - 2\sum_{k=1}^{p-1}(\overline{m}_k+5)H_{\overline{m}_k+2},\end{aligned}$$

where

- $\overline{m}_k = |\overline{I}_k| = \ell_{k+1} - u_k - 1$
- $m_k = |I_k| = u_k - \ell_k + 1$
- $m = m_1 + m_2 + \cdots + m_p$

Chunksort vs. quicksort+quickselect

- For small p ($p = 1, 2$) it is perfectly reasonable to solve the problem using quickselect to find the beginning and end of each block, and then sort each block using quicksort
- The order of magnitude of the average cost of chunksort and this alternative is similar; but there are significative diferences for the second order terms
- For example, if $p = 1$ and $I_1 = [\alpha \cdot n - f(n), \alpha \cdot n + f(n)]$ for some $\alpha < 1/2$ and $f(n) = o(n)$ then chunksort makes $2(1 - \alpha)n$ comparisons less

1. Introduction
2. The algorithm
3. The analysis
4. Conclusions

Conclusions

- The formula for the average cost of chunksort generalizes the corresponding formulas for special cases: quicksort, quickselect, partial quicksort, multiple quickselect, ...

Conclusions

- The formula for the average cost of chunksort generalizes the corresponding formulas for special cases: quicksort, quickselect, partial quicksort, multiple quickselect, . . .
- Despite being simple and "efficient", chunksort should not be used as a substitute for the specialized algorithms (maybe it could be used for the less frequent tasks of multiple selection or partial sorting)

Conclusions

- It is interesting to analyze the cost of the algorithm when taking into account the cost $\Theta(\log p)$ of locating the pivot's position in the array of intervals

Conclusions

- It is interesting to analyze the cost of the algorithm when taking into account the cost $\Theta(\log p)$ of locating the pivot's position in the array of intervals
- I would like to know about possible applications for chunksort; e.g., partial quicksort has been used to improve significantly the practical performance of Kruskal's algorithm for minimum spanning trees

ytTro hai nek taoy rnunotouf

Thank you for your attention