Interval Sorting

Conrado Martínez
U. Politècnica Catalunya

LIAFA, U. Paris 7, May 2010

Joint work with:

R.M. Jiménez

Introduction

The problem:

Input: An array $A[1..n]$ of n items drawn from a totally ordered domain; a set $I = \{[\ell_t, u_t] \,|\, 1 \leq t \leq p\}$ of p disjoint intervals with

$$1 \leq \ell_1 \leq u_1 < \ell_2 \leq u_2 < \cdots < \ell_p \leq u_p \leq n,$$

Output: The array A rearranged in such a way that

1. $A[\ell_t..u_t]$ contains the ℓ_t-th,...,u_t-th smallest elements of A in nondecreasing order, for all t, $1 \leq t \leq p$
2. $A[u_t + 1..\ell_{t+1} - 1]$ contains the $(u_t + 1)$-th, ..., $(\ell_{t+1} - 1)$-th smallest elements of A, for all t, $0 \leq t \leq p$ $(u_0 = 0$, $\ell_{p+1} = n + 1)$

Introduction

Example

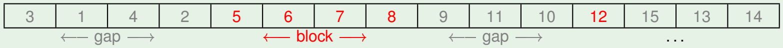
$p = 2$, $I_1 = [5, 8]$, $I_2 = [12, 12]$

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

Introduction

Example

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



Introduction

The main interest of interval sorting is that it generalizes several related fundamental problems:

- Sorting: $p = 1, I = \{[1, n]\}$
- Selection of the j-th: $p = 1, I = \{[j, j]\}$
- Multiple selection: $I = \{[j_1, j_1], [j_2, j_2], \ldots, [j_p, j_p]\}$
- Partial sorting: $p = 1, I = \{[1, m]\}, m < n$

Introduction

The main interest of interval sorting is that it generalizes several related fundamental problems:

- Sorting: $p = 1, I = \{[1, n]\}$
- Selection of the j-th: $p = 1, I = \{[j, j]\}$
- Multiple selection: $I = \{[j_1, j_1], [j_2, j_2], \ldots, [j_p, j_p]\}$
- Partial sorting: $p = 1, I = \{[1, m]\}, m < n$

Introduction

The main interest of interval sorting is that it generalizes several related fundamental problems:

- Sorting: $p = 1, I = \{[1, n]\}$
- Selection of the j-th: $p = 1, I = \{[j, j]\}$
- Multiple selection: $I = \{[j_1, j_1], [j_2, j_2], \ldots, [j_p, j_p]\}$
- Partial sorting: $p = 1, I = \{[1, m]\}, m < n$

Introduction

The main interest of interval sorting is that it generalizes several related fundamental problems:

- Sorting: $p = 1, I = \{[1, n]\}$
- Selection of the j-th: $p = 1, I = \{[j, j]\}$
- Multiple selection: $I = \{[j_1, j_1], [j_2, j_2], \dots, [j_p, j_p]\}$
- Partial sorting: $p = 1, I = \{[1, m]\}, m < n$

Introduction

- Other instances of interval sorting might be useful:
  - Sort & filter: $p = 1, I = [\beta n, (1-\beta)n], \beta < 1/2$
  - Outliers: $p = 2, I = \{[1, k], [n-k+1, n]\}$
- Sorting A in (expected) time $\Theta(n \log n)$ solves the problem, but this is wasteful if $m = |I_1| + \ldots + |I_p| \ll n$

Introduction

- Other instances of interval sorting might be useful:
  - Sort & filter: $p = 1, I = [\beta n, (1 - \beta)n], \beta < 1/2$
  - Outliers: $p = 2, I = \{[1, k], [n - k + 1, n]\}$
- Sorting A in (expected) time $\Theta(n \log n)$ solves the problem, but this is wasteful if $m = |I_1| + \ldots + |I_p| \ll n$

Introduction

- Other instances of interval sorting might be useful:
  - Sort & filter: $p = 1, I = [\beta n, (1-\beta)n], \beta < 1/2$
  - Outliers: $p = 2, I = \{[1,k],[n-k+1,n]\}$
- Sorting A in (expected) time $\Theta(n \log n)$ solves the problem, but this is wasteful if $m = |I_1| + \ldots + |I_p| \ll n$

Introduction

- Other instances of interval sorting might be useful:
  - Sort & filter: $p = 1, I = [\beta n, (1-\beta)n], \beta < 1/2$
  - Outliers: $p = 2, I = \{[1,k],[n-k+1,n]\}$
- Sorting A in (expected) time $\Theta(n \log n)$ solves the problem, but this is wasteful if $m = |I_1| + \ldots + |I_p| \ll n$

What's ahead?

What’s ahead?

1. Chunksort: A simple divide & conquer algorithm for interval sorting
2. Average performance of chunksort
3. A simple lower bound for interval sorting
4. Intermezzo:
    - Optimal sampling strategies for quicksort
    - Optimal sampling strategies for quickselect
5. “Optimal” chunksort
6. Disgression: How far from optimal?

What's ahead?

1. Chunksort: A simple divide & conquer algorithm for interval sorting
2. Average performance of chunksort
3. A simple lower bound for interval sorting
4. Intermezzo:
   - Optimal sampling strategies for quicksort
   - Optimal sampling strategies for quickselect
5. "Optimal" chunksort
6. Disgression: How far from optimal?

What’s ahead?

1. Chunksort: A simple divide & conquer algorithm for interval sorting
2. Average performance of chunksort
3. A simple lower bound for interval sorting
4. Intermezzo:
   - Optimal sampling strategies for quicksort
   - Optimal sampling strategies for quickselect
5. “Optimal” chunksort
6. Disgression: How far from optimal?

What's ahead?

1. Chunksort: A simple divide & conquer algorithm for interval sorting
2. Average performance of chunksort
3. A simple lower bound for interval sorting
4. Intermezzo:
    - Optimal sampling strategies for quicksort
    - Optimal sampling strategies for quickselect
5. "Optimal" chunksort
6. Disgression: How far from optimal?

What’s ahead?

1. Chunksort: A simple divide & conquer algorithm for interval sorting
2. Average performance of chunksort
3. A simple lower bound for interval sorting
4. Intermezzo:
    - Optimal sampling strategies for quicksort
    - Optimal sampling strategies for quickselect
5. “Optimal” chunksort
6. Disgression: How far from optimal?

What’s ahead?

1. Chunksort: A simple divide & conquer algorithm for interval sorting
2. Average performance of chunksort
3. A simple lower bound for interval sorting
4. Intermezzo:
   - Optimal sampling strategies for quicksort
   - Optimal sampling strategies for quickselect
5. “Optimal” chunksort
6. Disgression: How far from optimal?

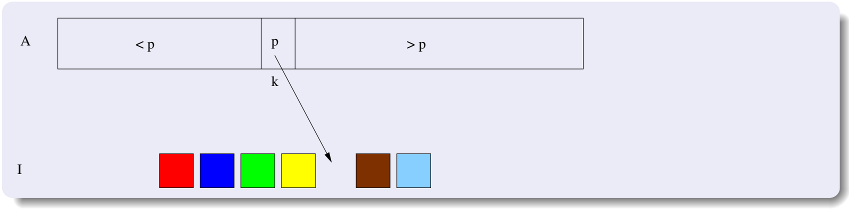
What's ahead?

1. Chunksort: A simple divide & conquer algorithm for interval sorting
2. Average performance of chunksort
3. A simple lower bound for interval sorting
4. Intermezzo:
    - Optimal sampling strategies for quicksort
    - Optimal sampling strategies for quickselect
5. "Optimal" chunksort
6. Disgression: How far from optimal?

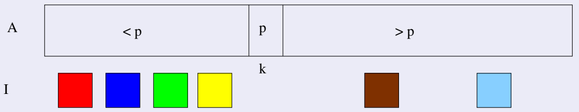
Chunksort

```
procedure CHUNKSORT($A$, $i$, $j$, $I$, $r$, $s$)
    if $i \geq j$ then return ▷ $A$ contains one or no elements
    if $r \leq s$ then
        $pv \leftarrow$ SELECTPIVOT($A, i, j$)
        PARTITION($A, pv, i, j, k$)
        $t \leftarrow$ LOCATE($I, r, s, k$)
▷ Locate the value $t$ such that $\ell_t \leq k \leq u_t$ with $I_t = [\ell_t, u_t]$,
▷ or $u_t < k < \ell_{t+1}$
        if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
            CHUNKSORT($A, i, k-1, I, r, t$)
            CHUNKSORT($A, k+1, j, I, t+1, s$)
        else ▷ $k$ falls in the $t$-th interval
            CHUNKSORT($A, i, k-1, I, r, t$)
            CHUNKSORT($A, k+1, j, I, t, s$)
```

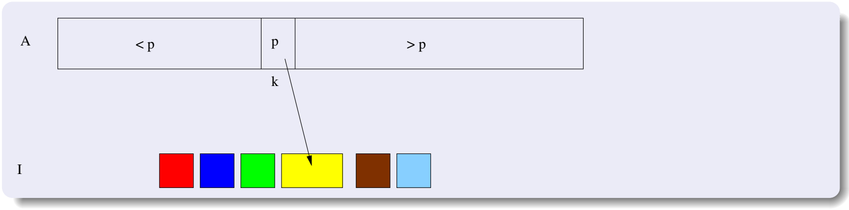
Chunksort: An example



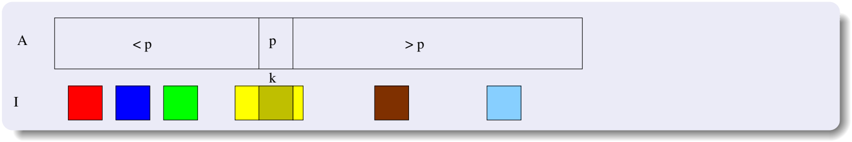
Chunksort: An example



Chunksort: An example



Chunksort: An example



Chunksort

Example (Using chunksort to sort)

- $p = 1$, $I_1 = [1, n]$
- $1 \le k \le n \implies \ell_1 \le k \le u_1 \implies r = s = t = 1$

```
procedure CHUNKSORT($A$, $i$, $j$, $I$, $r$, $s$)
    ...
    if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t+1, s$)
    else ▷ $k$ falls in the $t$-th interval
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t, s$)
```

Chunksort

Example (Using chunksort for selection)

- $p = 1$, $I_1 = [m, m]$
- $m < k \implies t = 1, u_1 < k$

```
procedure CHUNKSORT($A$, $i$, $j$, $I$, $r$, $s$)
    ...
    if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t+1, s$)
    else ▷ $k$ falls in the $t$-th interval
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t, s$)
```

Chunksort

Example (Using chunksort for selection)

- $p = 1$, $I_1 = [m, m]$
- $m < k \implies t = 1, u_1 < k$

```
procedure CHUNKSORT($A$, $i$, $j$, $I$, $r$, $s$)
    ...
    if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t+1, s$)
    else ▷ $k$ falls in the $t$-th interval
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t, s$)
```

Chunksort

Example (Using chunksort for selection)

- $p = 1$, $I_1 = [m, m]$
- $k < m \implies t = 0, u_0 < k < \ell_1$

```
procedure CHUNKSORT($A$, $i$, $j$, $I$, $r$, $s$)
    ...
    if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t+1, s$)
    else ▷ $k$ falls in the $t$-th interval
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t, s$)
```

Chunksort

Example (Using chunksort for selection)

- $p = 1$, $I_1 = [m, m]$
- $k < m \implies t = 0, u_0 < k < \ell_1$

```
procedure CHUNKSORT($A$, $i$, $j$, $I$, $r$, $s$)
    ...
    if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t+1, s$)
    else ▷ $k$ falls in the $t$-th interval
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t, s$)
```

Chunksort

Example (Using chunksort for partial sorting)

- $p = 1,\ I_1 = [1, m]$
- $1 \leq k \leq m \implies \ell_1 \leq k \leq u_1 \implies r = s = t = 1, k \leq u_1$

```
procedure CHUNKSORT($A, i, j, I, r, s$)
    . . .
    if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t+1, s$)
    else ▷ $k$ falls in the $t$-th interval
        CHUNKSORT($A, i, k-1, I, r, t$)
        CHUNKSORT($A, k+1, j, I, t, s$)
```

Chunksort

Example (Using chunksort for partial sorting)

- $p = 1,\ I_1 = [1, m]$
- $m < k \leq n \implies u_1 < k \leq \ell_2 \implies r = s = t = 1, u_1 < k$

```
procedure CHUNKSORT($A$, $i$, $j$, $I$, $r$, $s$)
    ...
    if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
        CHUNKSORT$(A, i, k-1, I, r, t)$
        CHUNKSORT$(A, k+1, j, I, t+1, s)$
    else ▷ $k$ falls in the $t$-th interval
        CHUNKSORT$(A, i, k-1, I, r, s)$
        CHUNKSORT$(A, k+1, j, I, t, s)$
```

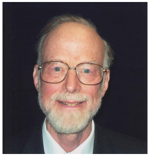
Chunksort

Example (Using chunksort for partial sorting)

- $p = 1,\ I_1 = [1, m]$
- $m < k \leq n \implies u_1 < k \leq \ell_2 \implies r = s = t = 1, u_1 < k$

```
procedure CHUNKSORT($A$, $i$, $j$, $I$, $r$, $s$)
    ...
    if $u_t < k$ then ▷ $k$ falls in the $t$-th gap
        CHUNKSORT$(A, i, k-1, I, r, t)$
        CHUNKSORT$(A, k+1, j, I, t+1, s)$
    else ▷ $k$ falls in the $t$-th interval
        CHUNKSORT$(A, i, k-1, I, r, s)$
        CHUNKSORT$(A, k+1, j, I, t, s)$
```

Quicksort: Average cost


C.A.R. Hoare

- Probability that the selected pivot is the k-th of n elements: $\pi_{n,k}$; for the basic variants here $\pi_{n,k} = 1/n$
- Average number of comparisons Q_n to sort n elements:

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

- 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: Average cost

D.E. Knuth

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

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

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

$$C_{n,m} = 2\big(n + 3 + (n+1)H_n \\ - (n+3-m)H_{n+1-m} - (m+2)H_m\big)$$

Partial quicksort: Average cost

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

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

- The solution is (Martínez, 2004):

$$\begin{aligned} P_{n,m} &= 2n + 2(n+1)H_n - 2(n+3-m)H_{n+1-m} \\ &\quad - 6m + 6 \end{aligned}$$

A Bit of Notation

- $I_t = [\ell_t, u_t]$: the t-th interval, $1 \leq t \leq p$
- $\overline{I}_t = [u_t + 1..\ell_{t+1} - 1]$: the t-th gap, $0 \leq t \leq p$
- $m_t = |I_t| = u_t - \ell_t + 1$: size of the t-th interval
- $\overline{m}_t = |\overline{I}_t| = \ell_{t+1} - u_t - 1$: size of the t-th gap
- $m = m_1 + \ldots + m_p$: # of elements to be sorted
- $\overline{m} = \overline{m}_0 + \ldots + \overline{m}_p = n - m$: # of elements not sorted

A Bit of Notation

- $I_t = [\ell_t, u_t]$: the t-th interval, $1 \leq t \leq p$
- $\overline{I}_t = [u_t + 1..\ell_{t+1} - 1]$: the t-th gap, $0 \leq t \leq p$
- $m_t = |I_t| = u_t - \ell_t + 1$: size of the t-th interval
- $\overline{m}_t = |\overline{I}_t| = \ell_{t+1} - u_t - 1$: size of the t-th gap
- $m = m_1 + \ldots + m_p$: # of elements to be sorted
- $\overline{m} = \overline{m}_0 + \ldots + \overline{m}_p = n - m$: # of elements not sorted

A Bit of Notation

- $I_t = [\ell_t, u_t]$: the t-th interval, $1 \le t \le p$
- $\overline{I}_t = [u_t + 1..\ell_{t+1} - 1]$: the t-th gap, $0 \le t \le p$
- $m_t = |I_t| = u_t - \ell_t + 1$: size of the t-th interval
- $\overline{m}_t = |\overline{I}_t| = \ell_{t+1} - u_t - 1$: size of the t-th gap
- $m = m_1 + \ldots + m_p$: # of elements to be sorted
- $\overline{m} = \overline{m}_0 + \ldots + \overline{m}_p = n - m$: # of elements not sorted

A Bit of Notation

- $I_t = [\ell_t, u_t]$: the t-th interval, $1 \le t \le p$
- $\overline{I}_t = [u_t + 1..\ell_{t+1} - 1]$: the t-th gap, $0 \le t \le p$
- $m_t = |I_t| = u_t - \ell_t + 1$: size of the t-th interval
- $\overline{m}_t = |\overline{I}_t| = \ell_{t+1} - u_t - 1$: size of the t-th gap
- $m = m_1 + \ldots + m_p$: # of elements to be sorted
- $\overline{m} = \overline{m}_0 + \ldots + \overline{m}_p = n - m$: # of elements not sorted

A Bit of Notation

- $I_t = [\ell_t, u_t]$: the t-th interval, $1 \leq t \leq p$
- $\overline{I}_t = [u_t + 1..\ell_{t+1} - 1]$: the t-th gap, $0 \leq t \leq p$
- $m_t = |I_t| = u_t - \ell_t + 1$: size of the t-th interval
- $\overline{m}_t = |\overline{I}_t| = \ell_{t+1} - u_t - 1$: size of the t-th gap
- $m = m_1 + \ldots + m_p$: # of elements to be sorted
- $\overline{m} = \overline{m}_0 + \ldots + \overline{m}_p = n - m$: # of elements not sorted

A Bit of Notation

- $I_t = [\ell_t, u_t]$: the t-th interval, $1 \leq t \leq p$
- $\overline{I}_t = [u_t + 1..\ell_{t+1} - 1]$: the t-th gap, $0 \leq t \leq p$
- $m_t = |I_t| = u_t - \ell_t + 1$: size of the t-th interval
- $\overline{m}_t = |\overline{I}_t| = \ell_{t+1} - u_t - 1$: size of the t-th gap
- $m = m_1 + \ldots + m_p$: # of elements to be sorted
- $\overline{m} = \overline{m}_0 + \ldots + \overline{m}_p = n - m$: # of elements not sorted

Chunksort: 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 ($\pi_{n,k} = 1/n$)
- $C_{n;\{I_r,\dots,I_s\}}$ = the average number of comparisons needed to do interval sort on n elements for the given set of intervals $\{I_r, \dots, I_s\}$

Chunksort: 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 ($\pi_{n,k} = 1/n$)
- $C_{n;\{I_r,\ldots,I_s\}}$ = the average number of comparisons needed to do interval sort on n elements for the given set of intervals $\{I_r, \ldots, I_s\}$

Chunksort: 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 ($\pi_{n,k} = 1/n$)
- $C_{n;\{I_r,\ldots,I_s\}}$ = the average number of comparisons needed to do interval sort on n elements for the given set of intervals $\{I_r, \ldots, I_s\}$

Chunksort: 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 ($\pi_{n,k} = 1/n$)
- $C_{n;\{I_r,\ldots,I_s\}}$ = the average number of comparisons needed to do interval sort on n elements for the given set of intervals $\{I_r, \ldots, I_s\}$

Chunksort: The recurrence

$$C_{n;\{I_r,\dots,I_s\}} = n-1+\sum_{t=r-1}^{s}\sum_{k\in\overline{I}_t}\pi_{n,k}\big(C_{k-1;\{I_r,\dots,I_t\}}+C_{n-k;\{I_{t+1},\dots,I_s\}}\big)$$
$$+\sum_{t=r}^{s}\sum_{k\in I_t}\pi_{n,k}\big(C_{k-1;\{I_r,\dots,I_t\}}+C_{n-k;\{I_t,\dots,I_s\}}\big),$$

How to solve the recurrence . . .

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

$$C(z; x, y) = \sum_{n \geq 0} \sum_{1 \leq i \leq j \leq n} C_{n;\{[i,j]\}} z^n \, x^i y^j,$$

  which is actually a first-order linear differential equation

How to solve the recurrence . . .

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

$$C(z; x, y) = \sum_{n\geq 0} \sum_{1\leq i\leq j\leq n} C_{n;\{[i,j]\}} z^n \, x^i y^j ,$$

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

$$C(z; x_1, y_1, x_2, y_2) = \sum_{n \geq 0} \sum_{1 \leq i \leq j \leq i' \leq j' \leq n} C_{n;\{[i,j],[i',j']\}} z^n \, x_1^i y_1^j x_2^{i'} y_2^{j'},$$

  which satisfies a similar ODE involving $C(z; \cdot, \cdot)$

- A pattern emerges here, so that one can obtain a general form for the functional equation satisfied by $C(z; u_1, v_1, \ldots, u_p, v_p)$
- Solve and extract the coefficients

How to solve the recurrence . . .

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

$$C(z; x_1, y_1, x_2, y_2) = \sum_{n \geq 0} \sum_{1 \leq i \leq j \leq i' \leq j' \leq n} C_{n;\{[i,j],[i',j']\}} z^n \, x_1^i y_1^j x_2^{i'} y_2^{j'},$$

  which satisfies a similar ODE involving $C(z; \cdot, \cdot)$
- A pattern emerges here, so that one can obtain a general form for the functional equation satisfied by $C(z; u_1, v_1, \ldots, u_p, v_p)$
- Solve and extract the coefficients

How to solve the recurrence . . .

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

  $$C(z; x_1, y_1, x_2, y_2) = \sum_{n \geq 0} \sum_{1 \leq i \leq j \leq i' \leq j' \leq n} C_{n;\{[i,j],[i',j']\}} z^n \, x_1^i y_1^j x_2^{i'} y_2^{j'},$$

  which satisfies a similar ODE involving $C(z; \cdot, \cdot)$
- A pattern emerges here, so that one can obtain a general form for the functional equation satisfied by $C(z; u_1, v_1, \dots, u_p, v_p)$
- Solve and extract the coefficients

…but how we actually did solve it

We guessed the solution from the known solutions for quicksort, quickselect, partial quicksort and multiple quickselect and proved it by induction…

Chunksort: Average cost

Theorem

The average number of element comparisons $C_n := C_{n;\{I_1,\dots,I_p\}}$ needed by chunksort given the intervals $\{I_1,\dots,I_p\}$ is

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

A simple lower bound for interval sorting

- $\Lambda(n, \mathbf{m}, \overline{\mathbf{m}})$ = minimum # of comparisons needed on average to solve interval sorting of intervals with sizes $\mathbf{m} = (m_1, \ldots, m_p)$ and gaps $\overline{\mathbf{m}} = (\overline{m}_0, \ldots, \overline{m}_p)$
- The two vectors $\mathbf{m}$, $\overline{\mathbf{m}}$ and the value n univocally determining the interval sorting instance
- Suppose we perform an optimal interval sort of the array of n elements, then we sort optimally the gaps; hence

$$\Lambda(n, \mathbf{m}, \overline{\mathbf{m}}) + \sum_{t=0}^{p} \log_2(\overline{m}_t!) \geq \log_2(n!)$$

A simple lower bound for interval sorting

- $\Lambda(n, \mathbf{m}, \overline{\mathbf{m}})$ = minimum # of comparisons needed on average to solve interval sorting of intervals with sizes $\mathbf{m} = (m_1, \ldots, m_p)$ and gaps $\overline{\mathbf{m}} = (\overline{m}_0, \ldots, \overline{m}_p)$
- The two vectors $\mathbf{m}$, $\overline{\mathbf{m}}$ and the value n univocally determining the interval sorting instance
- Suppose we perform an optimal interval sort of the array of n elements, then we sort optimally the gaps; hence

$$\Lambda(n, \mathbf{m}, \overline{\mathbf{m}}) + \sum_{t=0}^{p} \log_2(\overline{m}_t!) \geq \log_2(n!)$$

A simple lower bound for interval sorting

- $\Lambda(n, \mathbf{m}, \overline{\mathbf{m}})$ = minimum # of comparisons needed on average to solve interval sorting of intervals with sizes $\mathbf{m} = (m_1, \ldots, m_p)$ and gaps $\overline{\mathbf{m}} = (\overline{m}_0, \ldots, \overline{m}_p)$
- The two vectors $\mathbf{m}$, $\overline{\mathbf{m}}$ and the value n univocally determining the interval sorting instance
- Suppose we perform an optimal interval sort of the array of n elements, then we sort optimally the gaps; hence

$$\Lambda(n, \mathbf{m}, \overline{\mathbf{m}}) + \sum_{t=0}^{p} \log_2(\overline{m}_t!) \geq \log_2(n!)$$

A simple lower bound for interval sorting

Lemma

$$\Lambda(n, \mathbf{m}, \overline{\mathbf{m}}) \geq \sum_{t=1}^{p} m_t \log_2 m_t + n\mathcal{H}\left(\{\overline{m}_0/n, m_1/n, \overline{m}_1/n, \ldots, m_p/n, \overline{m}_p/n\}\right) - m \log_2 e + o(n)$$

with $\mathcal{H}(\{q_t\}) = -\sum_t q_t \log_2 q_t$ *denoting the entropy of the discrete probability distribution* $\{q_t\}$ *and* $m = m_1 + \ldots + m_p$*.*

Optimal quicksort

M. H. van Emden

- Using the median of a small sample as the pivot of each recursive call of quicksort improves the average cost of quicksort (Singleton's median-of-3, 1969)
- Van Emden (1970) and Hennequin (1989) have studied the performance of quicksort with median-of-$(2t+1)$ showing an steady improvement of performance

$$C_n^{(t)} = c_t n \log_2 n, \qquad c_0 = 2\ln 2 = 1.386, c_1 = 1.188, \ldots, c_\infty = 1$$

Optimal quicksort

M. H. van Emden

- Using the median of a small sample as the pivot of each recursive call of quicksort improves the average cost of quicksort (Singleton's median-of-3, 1969)
- Van Emden (1970) and Hennequin (1989) have studied the performance of quicksort with median-of-$(2t+1)$ showing an steady improvement of performance

$$C_n^{(t)} = c_t n \log_2 n, \qquad c_0 = 2\ln 2 = 1.386, c_1 = 1.188, \ldots, c_\infty = 1$$

Optimal quicksort

C. C. McGeoch

S. Roura

J.D. Tygar

- McGeoch and Tygar (1995) considered using the median of a variable-size sample for the first round, then fixed size samples on subsequent calls
- Martínez and Roura (2001) studied the use of variable-size sampling for quicksort and quickselect, showing that optimal expected performance can be achieved

Optimal quicksort

C. C. McGeoch

S. Roura

J.D. Tygar

- McGeoch and Tygar (1995) considered using the median of a variable-size sample for the first round, then fixed size samples on subsequent calls
- Martínez and Roura (2001) studied the use of variable-size sampling for quicksort and quickselect, showing that optimal expected performance can be achieved

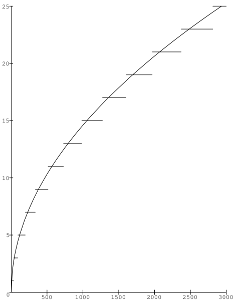
Optimal quicksort

Theorem (Martínez, Roura, 2001)

The expected performance of quicksort using as pivots the median of samples of size $s = s(n)$, such that $s \to \infty$ and $s/n \to 0$ as $n \to \infty$ is

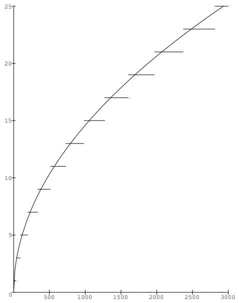
$$n \log_2 n + \textit{lower order terms}$$

Optimal quicksort



- The lower order terms are minimized by choosing samples of size $\Theta(\sqrt{n})$
- The constant hidden in $\Theta(\sqrt{n})$ depends on the (linear) time algorithm used to find the median of the samples

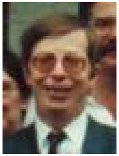
Optimal quicksort



- The lower order terms are minimized by choosing samples of size $\Theta(\sqrt{n})$
- The constant hidden in $\Theta(\sqrt{n})$ depends on the (linear) time algorithm used to find the median of the samples

Optimal quickselect

R. Grübel

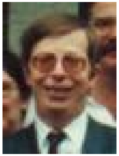

P. Kirschenhofer

H. Prodinger

- Median-of-$(2t+1)$ sampling can also be used for quickselect
- The improvements on the performance have been studied by several authors: Kirschenhofer, Prodinger, Martínez (1997), Grübel (1999), Martínez and Roura (2001)
- But . . . is the median of the sample a good choice?

Optimal quickselect

R. Grübel



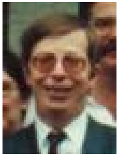
P. Kirschenhofer

H. Prodinger

- Median-of-$(2t + 1)$ sampling can also be used for quickselect
- The improvements on the performance have been studied by several authors: Kirschenhofer, Prodinger, Martínez (1997), Grübel (1999), Martínez and Roura (2001)
- But . . . is the median of the sample a good choice?

Optimal quickselect

R. Grübel


P. Kirschenhofer

H. Prodinger

- Median-of-$(2t+1)$ sampling can also be used for quickselect
- The improvements on the performance have been studied by several authors: Kirschenhofer, Prodinger, Martínez (1997), Grübel (1999), Martínez and Roura (2001)
- But . . . is the median of the sample a good choice?

Optimal quickselect

D. Panario

A. Viola

- In 2004, Martínez, Panario and Viola consider variants of quickselect where the rank r of the pivot within the sample of size s is proportional to the rank j of the sought element in the array n:

$$r \approx \frac{j}{n} \cdot s$$

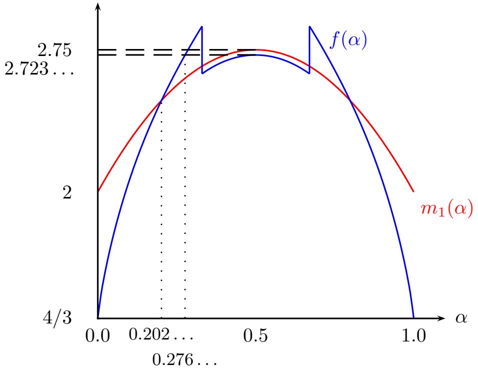
- More in general, they consider all variants where r is a function of $\alpha = j/n$

Optimal quickselect

- For all variants

$$C_{n,j} = f(\alpha) \cdot n + o(n), \alpha = j/n,$$

for instance, $f(\alpha) = 2 + 2\mathcal{H}(\alpha)$ for standard quickselect and $f(\alpha) = 2 + 3\alpha(1-\alpha)$ for median-of-three



Optimal quickselect

- Optimal expected performance can be achieve with 3 basic “ingredients:”
  - Using variable-sample sizes $s = s(n)$ with $s \to \infty$, $s/n \to 0$
  - The rank of the pivot withis the sample must be $r \sim \alpha \cdot s$
  - If the souhgt element has rank $j > n/2$ take $r = \alpha \cdot s - \delta$; if $j < n/2$ then $r = \alpha \cdot s + \delta$, for some “small” δ, say $\delta = \sqrt{s}$
  - You want the chosen pivot to land very close to j on the correct side with high probability

Optimal quickselect

- Optimal expected performance can be achieve with 3 basic “ingredients:”
    - Using variable-sample sizes $s = s(n)$ with $s \to \infty$, $s/n \to 0$
    - The rank of the pivot withis the sample must be $r \sim \alpha \cdot s$
    - If the souhgt element has rank $j > n/2$ take $r = \alpha \cdot s - \delta$; if $j < n/2$ then $r = \alpha \cdot s + \delta$, for some “small” δ, say $\delta = \sqrt{s}$
    - You want the chosen pivot to land very close to j on the correct side with high probability

Optimal quickselect

- Optimal expected performance can be achieve with 3 basic “ingredients:”
  - Using variable-sample sizes $s = s(n)$ with $s \to \infty$, $s/n \to 0$
  - The rank of the pivot withis the sample must be $r \sim \alpha \cdot s$
  - If the souhgt element has rank $j > n/2$ take $r = \alpha \cdot s - \delta$; if $j < n/2$ then $r = \alpha \cdot s + \delta$, for some “small” δ, say $\delta = \sqrt{s}$
  - You want the chosen pivot to land very close to j on the correct side with high probability

Optimal quickselect

- Optimal expected performance can be achieve with 3 basic “ingredients:”
  - Using variable-sample sizes $s = s(n)$ with $s \to \infty$, $s/n \to 0$
  - The rank of the pivot withis the sample must be $r \sim \alpha \cdot s$
  - If the souhgt element has rank $j > n/2$ take $r = \alpha \cdot s - \delta$; if $j < n/2$ then $r = \alpha \cdot s + \delta$, for some “small” δ, say $\delta = \sqrt{s}$
  - You want the chosen pivot to land very close to j on the correct side with high probability

Optimal quickselect

- Optimal expected performance can be achieve with 3 basic “ingredients:”
  - Using variable-sample sizes $s = s(n)$ with $s \to \infty$, $s/n \to 0$
  - The rank of the pivot withis the sample must be $r \sim \alpha \cdot s$
  - If the souhgt element has rank $j > n/2$ take $r = \alpha \cdot s - \delta$; if $j < n/2$ then $r = \alpha \cdot s + \delta$, for some “small” δ, say $\delta = \sqrt{s}$
  - You want the chosen pivot to land very close to j on the correct side with high probability

Optimal quickselect

Theorem (Martínez, Panario, Viola, 2004)

Any variant of quickselect using biased proportion-from-s with variable-size sampling has

$$f(\alpha) = 1 + \min(\alpha, 1 - \alpha)$$

Thus $C_{n,j} \sim n + \min(j, n - j) +$ *lower order terms*

Optimal chunksort

The recipy for optimality:

1. Merge small gaps: replace two intervals separated by a gap of size $o(n)$ by a single interval
2. If there is only one interval to sort and it contains $m = n - o(n)$ elements pick a pivot whose rank is close to $n/2$; use the median of a large ($\sqrt{n}$) sample
3. If not, choose some endpoint $\ell_r, u_r, \ldots, \ell_s, u_s$, say ρ
   - If $\rho = \ell_i$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the left of ρ
     If $\rho = u_i$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the right of ρ

Optimal chunksort

The recipy for optimality:

1. Merge small gaps: replace two intervals separated by a gap of size $o(n)$ by a single interval
2. If there is only one interval to sort and it contains $m = n - o(n)$ elements pick a pivot whose rank is close to $n/2$; use the median of a large ($\sqrt{n}$) sample
3. If not, choose some endpoint $\ell_r, u_r, \ldots, \ell_s, u_s$, say ρ
   - If $\rho = \ell_t$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the left of ρ
   - If $\rho = u_t$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the right of ρ

Optimal chunksort

The recipy for optimality:

1. Merge small gaps: replace two intervals separated by a gap of size $o(n)$ by a single interval
2. If there is only one interval to sort and it contains $m = n - o(n)$ elements pick a pivot whose rank is close to $n/2$; use the median of a large ($\sqrt{n}$) sample
3. If not, choose some endpoint $\ell_r, u_r, \ldots, \ell_s, u_s$, say ρ
    - If $\rho = \ell_t$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the left of ρ
    - If $\rho = u_t$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the right of ρ

Optimal chunksort

The recipy for optimality:

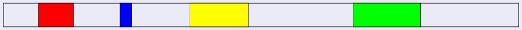
1. Merge small gaps: replace two intervals separated by a gap of size $o(n)$ by a single interval
2. If there is only one interval to sort and it contains $m = n - o(n)$ elements pick a pivot whose rank is close to $n/2$; use the median of a large ($\sqrt{n}$) sample
3. If not, choose some endpoint $\ell_r, u_r, \ldots, \ell_s, u_s$, say ρ
   - If $\rho = \ell_t$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the left of ρ
   - If $\rho = u_t$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the right of ρ

Optimal chunksort

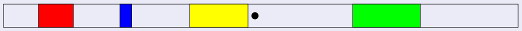
The recipy for optimality:

1. Merge small gaps: replace two intervals separated by a gap of size $o(n)$ by a single interval
2. If there is only one interval to sort and it contains $m = n - o(n)$ elements pick a pivot whose rank is close to $n/2$; use the median of a large ($\sqrt{n}$) sample
3. If not, choose some endpoint $\ell_r, u_r, \ldots, \ell_s, u_s$, say ρ
   - If $\rho = \ell_t$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the left of ρ
   - If $\rho = u_t$, pick a pivot from a large sample with rank proportional to ρ and biased to land to the right of ρ

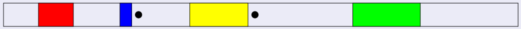
Optimal chunksort



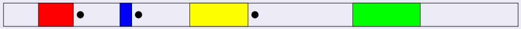
Optimal chunksort



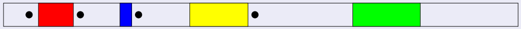
Optimal chunksort



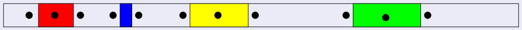
Optimal chunksort



Optimal chunksort



Optimal chunksort



Optimal chunksort

- The problem is thus to find the optimal order $\Longrightarrow$ dynamic programming
- Given the collection of endpoints $\rho_i = u_{r-1}$, $\rho_{i+1} = \ell_r, \ldots,$ $\rho_{j-1} = u_s$, $\rho_j = \ell_{s+1}$ find the endpoint ρ_k such that minimizes $c(i,j)$:

$$c(i,j) = \rho_j - \rho_i + \min_{i<k<j} (c(i,k) + c(k,j))$$

Optimal chunksort

- The problem is thus to find the optimal order $\Longrightarrow$ dynamic programming
- Given the collection of endpoints $\rho_i = u_{r-1}$, $\rho_{i+1} = \ell_r, \ldots,$ $\rho_{j-1} = u_s$, $\rho_j = \ell_{s+1}$ find the endpoint ρ_k such that minimizes $c(i,j)$:

$$c(i,j) = \rho_j - \rho_i + \min_{i<k<j} (c(i,k) + c(k,j))$$

Optimal chunksort

F.F. Yao

- The dynamic programming to find the optimal order to "cut the bar" has cost $O(p^3)$; it is almost analogous to building an optimal search tree where the weights of the leaves are the sizes of the intervals
- The efficiency of the algorithm can be greatly improved to $O(p^2)$ using Knuth-Yao's technique

Optimal chunksort

- We can use some heuristic to find a near-optimal solution to the “cut the bar” problem with cost $O(p \log p)$
- For instance, at each step, we can choose the endpoint ℓ_k or u_k which is closer to $(\rho_j - \rho_i)/2$; some care must be taken if we have ties, e.g., if $\ell_k = u_k$
- The analysis of the heuristic provides a useful upper bound on $c(0, 2p+1)$, the optimal cost of the “cut the bar” phase
- The total cost of chunksort becomes

$$\sum_{t=1}^{p} m_t \log_2 m_t + c(0, 2p+1) + O(p\sqrt{n})$$

$$\leq \sum_{t=1}^{p} m_t \log_2 m_t + n \cdot H + n + \text{lower order terms}$$

Optimal chunksort

- We can use some heuristic to find a near-optimal solution to the "cut the bar" problem with cost $O(p \log p)$
- For instance, at each step, we can choose the endpoint ℓ_k or u_k which is closer to $(\rho_j - \rho_i)/2$; some care must be taken if we have ties, e.g., if $\ell_k = u_k$
- The analysis of the heuristic provides a useful upper bound on $c(0, 2p+1)$, the optimal cost of the "cut the bar" phase
- The total cost of chunksort becomes

$$\sum_{t=1}^{p} m_t \log_2 m_t + c(0, 2p+1) + O(p\sqrt{n})$$

$$\leq \sum_{t=1}^{p} m_t \log_2 m_t + n \cdot H + n + \text{lower order terms}$$

Optimal chunksort

- We can use some heuristic to find a near-optimal solution to the "cut the bar" problem with cost $O(p \log p)$
- For instance, at each step, we can choose the endpoint ℓ_k or u_k which is closer to $(\rho_j - \rho_i)/2$; some care must be taken if we have ties, e.g., if $\ell_k = u_k$
- The analysis of the heuristic provides a useful upper bound on $c(0, 2p+1)$, the optimal cost of the "cut the bar" phase
- The total cost of chunksort becomes

$$\sum_{t=1}^{p} m_t \log_2 m_t + c(0, 2p+1) + O(p\sqrt{n})$$

$$\leq \sum_{t=1}^{p} m_t \log_2 m_t + n \cdot H + n + \text{lower order terms}$$

Optimal chunksort

- We can use some heuristic to find a near-optimal solution to the “cut the bar” problem with cost $O(p \log p)$
- For instance, at each step, we can choose the endpoint ℓ_k or u_k which is closer to $(\rho_j - \rho_i)/2$; some care must be taken if we have ties, e.g., if $\ell_k = u_k$
- The analysis of the heuristic provides a useful upper bound on $c(0, 2p+1)$, the optimal cost of the “cut the bar” phase
- The total cost of chunksort becomes

$$\begin{aligned}&\sum_{t=1}^{p} m_t \log_2 m_t + c(0, 2p+1) + O(p\sqrt{n}) \\ &\qquad \leq \sum_{t=1}^{p} m_t \log_2 m_t + n \cdot H + n + \text{lower order terms}\end{aligned}$$

Optimal chunksort

- Together with the lower bound for Λ

$$\sum_{t=1}^{p} m_t \log_2 m_t + n \cdot H - m \log_2 e + o(n) \leq \Lambda(n, \mathbf{m}, \overline{\mathbf{m}})$$
$$\leq \sum_{t=1}^{p} m_t \log_2 m_t + c(0, 2p+1) + O(p\sqrt{n})$$
$$\leq \sum_{t=1}^{p} m_t \log_2 m_t + n \cdot H + n + \text{lower order terms.}$$

- The lower and upper bounds differ by $n + o(n)$ comparisons if $p \ll \sqrt{n}$ (which indeed is the case, as we collapsed all “small” gaps!)

Optimal chunksort

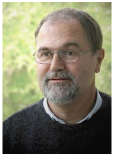
- Together with the lower bound for Λ

$$\begin{aligned}\sum_{t=1}^{p} m_t \log_2 m_t + n \cdot H - m \log_2 e + o(n) &\le \Lambda(n, \mathbf{m}, \overline{\mathbf{m}}) \\ &\le \sum_{t=1}^{p} m_t \log_2 m_t + c(0, 2p+1) + O(p\sqrt{n}) \\ &\le \sum_{t=1}^{p} m_t \log_2 m_t + n \cdot H + n + \text{lower order terms.}\end{aligned}$$

- The lower and upper bounds differ by $n + o(n)$ comparisons if $p \ll \sqrt{n}$ (which indeed is the case, as we collapsed all “small” gaps!)

Optimal chunksort

K. Kaligosi


K. Mehlhorn

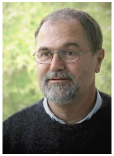
J. I. Munro

P. Sanders

- Kaligosi, Mehlhorn, Munro and Sanders (2005) have considered optimal multiple selection; they use similar techniques, but they propose an algorithm which picks a pivot close to the median for each recursive stage
- This yields a solution (for multiple selection) which is off by $O(n)$ comparisons from the optimal; our solution —which generalizes multiple selection— is off by at most $n + o(n)$ comparisons

Optimal chunksort

K. Kaligosi


K. Mehlhorn

J. I. Munro

P. Sanders

- Kaligosi, Mehlhorn, Munro and Sanders (2005) have considered optimal multiple selection; they use similar techniques, but they propose an algorithm which picks a pivot close to the median for each recursive stage
- This yields a solution (for multiple selection) which is off by $O(n)$ comparisons from the optimal; our solution —which generalizes multiple selection— is off by at most $n + o(n)$ comparisons

How far from optimal

1. The lower bound for $\Lambda(n, \mathrm{m}, \overline{\mathrm{m}})$ is not tight, for instance, for selection

$$\begin{aligned}\Lambda(n, \langle 1\rangle, \langle j-1, n-j\rangle) &= n+\min(j-1, n-j)+\text{l.o.t.} \quad \leftarrow \text{on avg!}\\ &\gg n\mathcal{H}\left(\{(j-1)/n, 1/n, (n-j)/n\}\right) + \text{l.o.t.}\end{aligned}$$

2. The upper bound corresponds to the heuristic for “cutting the bar”, and isn’t tight either

How far from optimal

1. The lower bound for $\Lambda(n, \mathrm{m}, \overline{\mathrm{m}})$ is not tight, for instance, for selection

$$
\begin{aligned}
\Lambda(n, \langle 1 \rangle, \langle j-1, n-j \rangle) &= n + \min(j-1, n-j) + \text{l.o.t.} \quad \leftarrow \text{on avg!} \\
&\gg n\mathcal{H}\left(\{(j-1)/n, 1/n, (n-j)/n\}\right) + \text{l.o.t.}
\end{aligned}
$$

2. The upper bound corresponds to the heuristic for “cutting the bar”, and isn’t tight either

How far from optimal

- The algorithm that we propose optimally solves sorting and selection
- We conjecture that it is optimal up to $o(n)$ comparisons for all interval sort instances, not just sorting and selection

How far from optimal

- The algorithm that we propose optimally solves sorting and selection
- We conjecture that it is optimal up to $o(n)$ comparisons for all interval sort instances, not just sorting and selection

Conclusions

- Interval sort's main interest is that it smoothly generalizes several fundamental problems: sorting, selection, multiple selection and partial sorting
- Chunksort (its basic variant) is a simple and elegant algorithm in the spirit of quicksort; its average performance is $\leq 2 + 2\ln 2 = 3.386$ times the optimal

Conclusions

- Interval sort's main interest is that it smoothly generalizes several fundamental problems: sorting, selection, multiple selection and partial sorting
- Chunksort (its basic variant) is a simple and elegant algorithm in the spirit of quicksort; its average performance is $\leq 2 + 2 \ln 2 = 3.386$ times the optimal

Conclusions

- Carefully choosing the pivots yields near-optimal performance; we conjecture it is optimal up to $o(n)$ comparisons
- For the choice of pivots we need to “orchestrate” two ingredients:
  - large samples and proportion-from to choose pivots landing near the places where we need them
  - dynamic programming/heuristic to find the optimal order to “cut the bar”

Conclusions

- Carefully choosing the pivots yields near-optimal performance; we conjecture it is optimal up to $o(n)$ comparisons
- For the choice of pivots we need to “orchestrate” two ingredients:
  - large samples and proportion-from to choose pivots landing near the places where we need them
  - dynamic programming/heuristic to find the optimal order to “cut the bar”

Conclusions

- Carefully choosing the pivots yields near-optimal performance; we conjecture it is optimal up to $o(n)$ comparisons
- For the choice of pivots we need to “orchestrate” two ingredients:
  - large samples and proportion-from to choose pivots landing near the places where we need them
  - dynamic programming/heuristic to find the optimal order to “cut the bar”

Conclusions

- Carefully choosing the pivots yields near-optimal performance; we conjecture it is optimal up to $o(n)$ comparisons
- For the choice of pivots we need to “orchestrate” two ingredients:
  - large samples and proportion-from to choose pivots landing near the places where we need them
  - dynamic programming/heuristic to find the optimal order to “cut the bar”

Conclusions

- There are several open problems remaining:
  - Better lower bounds
  - Proving the conjecture
  - Other randomized or deterministic algorithms
  - . . .

Conclusions

- There are several open problems remaining:
  - Better lower bounds
  - Proving the conjecture
  - Other randomized or deterministic algorithms
  - ...

Conclusions

- There are several open problems remaining:
  - Better lower bounds
  - Proving the conjecture
  - Other randomized or deterministic algorithms
  - . . .

Conclusions

- There are several open problems remaining:
  - Better lower bounds
  - Proving the conjecture
  - Other randomized or deterministic algorithms
  - . . .

purea icc!uMoboe

Merci beaucoup!