

Algorithmns and Datastructures

Cache Efficiency, Divide and Conquer

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science
Algorithmns and Datastructures, March 2016

Cache Efficiency

- Introduction

- Cache Organization

Divide and Conquer

- Introduction

Background:

- Up to now we always counted **number of operations**
- Assuming this is a good measure for the runtime of a algorithm/tool
- Today we will see examples where this is not suitable

Example:

- We sum up all elements of a field a of size n in ...
 - natural order:

$$\text{sum}(a) = a[1] + a[2] + \dots + a[n]$$

- random order:

$$\text{sum}(a) = a[21] + a[5] + \dots + a[8]$$

Python:

```
def init(size):  
    # use system time as seed  
    random.seed(None)  
  
    # set linear order as accessor  
    order = [a for a in range(0, size)]  
  
    # init array with random data  
    data = [random.random() for a in order]  
  
    return (order, data)
```

Python:

```
def run(param):  
    # unpack data  
    (order, data) = param  
  
    # init the sum value  
    s = 0  
  
    for index in order:  
        s += data[index]  
  
    return s
```

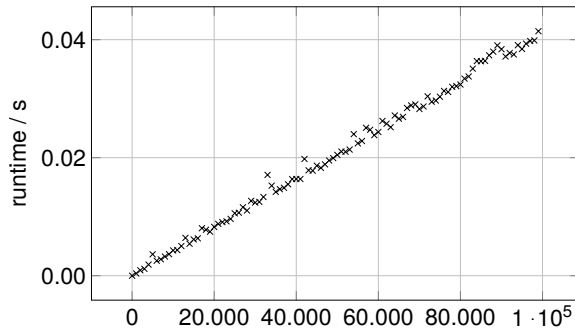


Figure: Summing elements in linear order

Python:

```
def init(size):  
    # use system time as seed  
    random.seed(None)  
  
    # set random order as accessor  
    order = [a for a in range(0, size)]  
    random.shuffle(order)  
  
    # init array with random data  
    data = [random.random() for a in order]  
  
    return (order, data)
```


Cache Efficiency

Random Order



Figure: Summing elements in random order

Conclusion:

- The number of operations are identical for both algorithms
- Accessing elements in random order takes a lot longer (Factor 10)

Why?

- The costs in terms of memory access are very different



Principle / organization:

- Accessing one byte of the main memory takes ≈ 100 ns
- Accessing one byte of (L1-)cache takes ≈ 1 ns
- Accessing one or more byte/s of main memory loads a whole block ≈ 100 B into the cache
- As long as this block is in the cache, it is not necessary to access the memory for bytes of this block



Cache organization:

- The (L1-)cache can hold multiple memory blocks (cache lines)
 - $\approx 100\text{kB}$
- If the capacity is reached unused blocks are discarded
 - Least recently used (LRU)
 - Least frequently used (LFU)
 - First in first out (FIFO)
- Details of discarding are not the topic for today



Terminology:

- The system consists of slow and fast memory
- The **slow memory** is divided in **blocks of size B**
- The **fast cache** has size M and can store M/B blocks
- If data is not in fast memory, the corresponding block is loaded into the **cache**



Terminology:

- The program defines which blocks are held in the **cache**
- We use the number of **block operations** as runtime estimation
- We ignore runtime costs of cache accesses / management

Block Operations



Figure: Comparison good / bad locality

Accessing the cache B times:

- **Best case:** 1 block operation \rightarrow good locality
- **Worst case:** B block operations \rightarrow bad locality

Additional factors:

- The following settings change only a small constant factor in number of block operations
 - The partitioning of the slow memory into blocks
 - If the block is 1 Bytes or 4 Bytes or 8 Bytes

Note:

- If the input size is smaller than M we load the complete data chunk directly into the cache
- Cache handling is only interesting when the input size is greater than M

Typical values: (Intel© i7-4770 Haswell, WD© Blue 2TB)

- CPU L1 Cache: $B = 64\text{ B}$, $M = 4 \times (32\text{ kB} + 32\text{ kB})$
- CPU L2 Cache: $B = 64\text{ B}$, $M = 4 \times 256\text{ kB}$
- CPU L3 Cache: $B = 64\text{ B}$, $M = 8\text{ MB}$
- Disk Cache: $B = 64\text{ kB}$, $M = 64\text{ MB}$
 - Many operating systems use free system memory as disk cache

Terminology:

- Block loads on CPU-cache are called **cache misses**
- Block operations on disk-cache are called **IOs**
(input / output operations)
- These also fall under the term **cache efficiency** or **IO efficiency**

Example 1 - Linear order:

- We sum up all elements in **natural order**

$$\text{sum}(a) = a[1] + a[2] + \dots + a[n]$$

- The number of block operations is $\text{ceil}(\frac{n}{B})$



Figure: Good locality of sum operation

Example 2 - Random order:

- We sum up all elements in **random order**

$$\text{sum}(a) = a[21] + a[5] + \dots + a[8]$$

- The number of block operations is n in the **worst case**
- This leads to a runtime factor difference of B



Figure: Bad locality of sum operation

Generally the factor is substantially $< B$

- Even with a **random order** we access per element 4 (int) / neighboring bytes at once
- If **not $n \gg M$** the next element might already with a high probability loaded in cache

QuickSort:

- **Strategy:** Divide and conquer
- Divide the data into two parts where the “left” part contains all values \leq those in the right part
- Choose one element (e.g the first one) as “pivot”-element
- Ideally both parts are the same size
- Both parts are sorted recursively



Figure: QuickSort with pivot-element

- **at start:** pivot in first position, first re-arrange list such that left part contains small, right part larger elements
- do required changes *in place*



- **end point:** k is left to left-most element greater than pivot
swap position 0 (pivot) with k (smaller than pivot)

Python:

```
def quicksort(l, start, end):  
    if (end - start) < 1:  
        return  
  
    i = start  
    k = end  
    piv = l[0]  
  
    ...
```



```
def quicksort(l, start, end):  
    ...  
  
    while k > i:  
        while l[i] <= piv and i <= end and k > i:  
            i += 1  
        while l[k] > piv and k >= start and k >= i:  
            k -= 1  
  
        if k > i: # swap elements  
            (l[i], l[k]) = (l[k], l[i])  
  
    (l[start], l[k]) = (l[k], l[start])  
    quicksort(l, start, k - 1)  
    quicksort(l, k + 1, end)
```

Number of operations for Quicksort:

- Let $T(n)$ be the runtime for the input size n
- Assumptions:
 - Fields are always separated perfectly in the middle
 - n is a power of two and recursion depth is $k = \log_2 n$

$$\begin{aligned} T(n) &\leq \underbrace{A \cdot n}_{\text{splitting in two parts}} + \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{recursive sort}} \\ &\leq A \cdot n + 2 \left(A \cdot \frac{n}{2} + 2 \cdot T\left(\frac{n}{4}\right) \right) \\ &= 2A \cdot n + 4 \cdot T\left(\frac{n}{4}\right) \\ &\leq 3A \cdot n + 8 \cdot T\left(\frac{n}{8}\right) \\ &\leq \dots \\ &\leq k \cdot A \cdot n + 2^k \cdot T(1) \\ &= \log_2 n \cdot A \cdot n + n \cdot T(1) \\ &\leq \log_2 n \cdot A \cdot n + n \cdot A \in \mathcal{O}(n \log_2 n) \end{aligned}$$



Figure: Locality of quicksort

- Let $IO(n)$ be the number of **block operations** for input size n
- Assumptions as before but recursion depth is $k = \log_2 \frac{n}{B}$
Why?

$$\begin{aligned} IO(n) &\leq \underbrace{A \cdot n/B}_{\text{splitting in two parts}} + \underbrace{2 \cdot IO(n/2)}_{\text{recursive sort}} \\ &\leq A \cdot n/B + 2(A \cdot n/2B + 2 \cdot IO(n/4)) \\ &\leq 2 \cdot A \cdot n/B + 4 \cdot IO(n/4) \\ &\leq 3 \cdot A \cdot n/B + 8 \cdot IO(n/8) \\ &\leq \dots \\ &\leq k \cdot A \cdot n/B + 2^k \cdot IO(n/2^k) \\ &= \log_2(n/B) \cdot A \cdot (n/B) + n/B \cdot IO(B) \\ &\leq \log_2(n/B) \cdot A \cdot (n/B) + A \cdot n/B \in O\left(\frac{n}{B} \cdot \log_2\left(\frac{n}{B}\right)\right) \end{aligned}$$

Concept:

- **Divide** the problem into smaller subproblems
- **Conquer** the subproblems through recursive solving.
If subproblems are small enough solve them directly
- **Connect** all solutions of the subproblems to a solution of the full problem
- **Recursive** application of the algorithm to ever smaller subproblems
- **Direct** solving of sufficiently small subproblems

- Function `solve` for solving a problem of size n

```
def solve(problem):  
    if n < threshold:  
        # solve directly  
        return solution  
    else:  
        # divide problem into subproblems  
        # P1, P2, ..., Pk with k>=2  
        S1 = solve(P1)  
        S2 = solve(P2)  
        ...  
        Sk = solve(Pk)  
  
        # combine solutions  
    return S1 + S2 + ... + Sk
```

- Can help with conceptual hard problems
 - **Solution** of the trivial problems has to be known
 - **Dividing** in subproblems has to be possible
 - **Combination** of solutions has to be possible
- Realization of **efficient solutions**
 - If trivial solution is $\in O(1)$
 - And separation / combination of subproblems is $\in O(n)$
 - And the number of subproblems is limited
 - The runtime is $\in O(n \cdot \log n)$
- Suitable for parallel processing
 - Subproblems are **independent** of each other
 - Only needed input for each subproblem has to be known

Definition of the trivial case:

- Smaller subproblems are elegant and simple
- Otherwise the efficiency will be improved if relative big subproblems can be solved directly
- Recursion depth should not get too big (stack / memory overhead)

Division in subproblems:

- Choosing the number of subproblems and the concrete allocation can be demanding

Combination of solutions:

- Typically conceptual demanding

Example - Maximum Subtotal Input:

- Progression X of n integers

Output:

- Maximum sum of related subsequence and its index boundary

Index	0	1	2	3	4	5	6	7	8	9
Value	31	-41	59	26	-53	58	97	-93	-23	84

Output: Sum: 187, Start: 2, End: 6

Application:

- Maximum profit of buying and selling shares



Naive solution (brute force)

```
def maxSubArray(X):  
    # Store sum, start, end  
    result = (X[0], 0, 0)  
    for i in range(0, len(X)):  
        for j in range(i, len(X)):  
            subSum = 0  
            for k in range(i, j + 1):  
                subSum += X[k]  
            if result[0] < subSum:  
                result = (subSum, i, j)  
    return result
```

Runtime - Upper bound

```
def maxSubArray(X):  
    result = (X[0], 0, 0)  
    # n loops -> O(n)  
    for i in range(0, len(X)):  
        # max n loops -> O(n)  
        for j in range(i, len(X)):  
            # max n loops -> O(n)  
            subSum = sum(X[i:j+1])  
            if result[0] < subSum: # O(1)  
                result = (subSum, i, j)  
    return result
```

Upper bound:

- Three interleaved loops
- Each loop with runtime $O(n)$
- Algorithm runtime of $O(n^3)$

Lower bound:

Table: Operations

i	Additions	j
$\frac{n}{3} \in O(n)$	$\frac{n}{3} \in O(n)$	$\frac{n}{3} \in O(n)$

- We iterate at least $\frac{n}{3}$ values for i
- For each i we iterate at least $\frac{n}{3}$ values for j
- For each j we have at least $\frac{n}{3}$ additions
- We need at least $T(n) = (\frac{n}{3})^3 \in \Omega(n^3)$ steps

Runtime:

- With $T(n) \in O(n^3)$ and $T(n) \in \Omega(n^3)$ we know:

$$T(n) \in \Theta(n^3)$$

- It is hard to solve the problem in a worse way ...

Current approach:

- Calculating the sum for range from i to j with loop

$$S_{i,j} = X[i] + X[i+1] + \dots + X[j]$$

Better approach:

- Incremental sum instead of loop

$$S_{i,j+1} = X[i] + X[i+1] + \dots + X[j] + X[j+1]$$

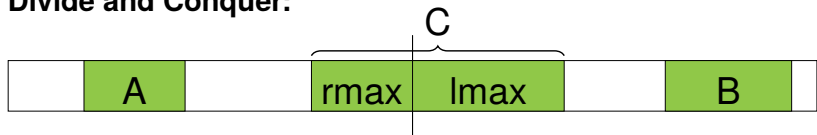
$$S_{i,j+1} = S_{i,j} + X[j+1] \in O(1) \quad \text{instead of} \quad \in O(n)$$

Better solution:

```
def maxSubArray(X):  
    result = (X[0], 0, 0)  
    # n loops -> O(n)  
    for i in range(0, len(X)):  
        subSum = 0  
        # max n loops -> O(n)  
        for j in range(i, len(X)):  
            subSum += X[j] # O(1)  
            if result[0] < subSum: # O(1)  
                result = (subSum, i, j)  
    return result
```

■ Runtime $\in O(n^2)$

Divide and Conquer:



Divide and Conquer Idea to solve:

- split the sequence in the middle
- Solve the left half of the problem
- Solve right half and combine both solutions into a total solution
- OK if maximum is located in **left half (A)** or **right half (B)**
- Problem: Maximum can **overlap split**
- To solve this case we have to calculate *rmax* and *lmax*
- The overall solution is the **maximum of A, B and C**

Principle - Divide and Conquer:

- Small problems are solved directly: $n = 1 \Rightarrow \text{max} = X[0]$
- Bigger problems are partitioned into two subproblems and recursively solved. Subsolutions A and B are returned.
- To determine subsolution C, rmax and lmax for the subproblems are computed.
- The overall solution is the **maximum of A, B and C**

Divide and conquer solution

```
def maxSubArray(X, i, j):  
    if i == j: #trivial case  
        return (X[i], i, i)  
    m = (i + j) / 2  
    #recursive Subsolutions for A,B  
    A = maxSubArray(X, i, m)  
    B = maxSubArray(X, m + 1, j)  
    #rmax and lmax for bordercase C  
    C1 = rmax(X, i, m)  
    C2 = lmax(X, m + 1, j)  
    C = (C1[0] + C2[0], C1[1], C2[1])  
    #Solution results from A,B,C  
    return max([A, B, C], \  
               key=lambda item: item[0])
```

■ General

[CRL01] Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson.

Introduction to Algorithms.

MIT Press, Cambridge, Mass, 2001.

[MS08] Kurt Mehlhorn and Peter Sanders.

Algorithms and data structures, 2008.

<https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Mehlhorn-Sanders-Toolbox.pdf>.

■ Caching

[Wik] [Cache](#)

`https://en.wikipedia.org/wiki/Cache`