Algorithms and Datastructures

Cache Efficiency, Divide and Conquer

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science

Algorithms and Datastructures, March 2018

Structure

Cache Efficiency

Introduction

Background:

Background:

- Up to now we always counted the number of operations
- Assuming this is a good measure for the runtime of a algorithm/tool

Background:

- Up to now we always counted the 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:

$$\mathrm{sum}(a) = a[1] + a[2] + \cdots + a[n]$$

  - random order:

$$\mathrm{sum}(a) = a[21] + a[5] + \cdots + a[8]$$

Cache Efficiency

Linear Order - Python

Python:

```python
def init(size):
    """Creates the dataset."""

    # 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)
```

Cache Efficiency

Linear Order - Python

Python:

```
def run(param):
    """Processes the dataset."""

    # unpack data
    (order, data) = param

    # init the sum value
    s = 0

    for index in order:
        s += data[index]

    return s
```

Cache Efficiency
Linear Order

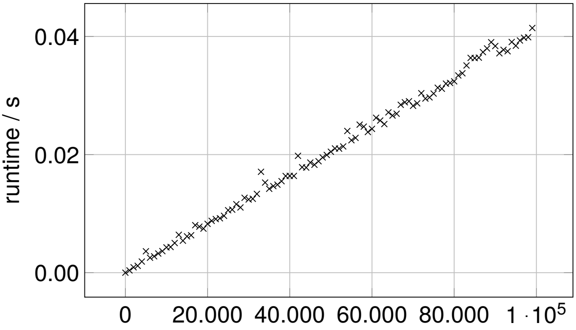


Figure: Summing elements in linear order

Cache Efficiency

Random Order - Python

```python
def init(size):
    """Creates a randomly ordered dataset."""

    # 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

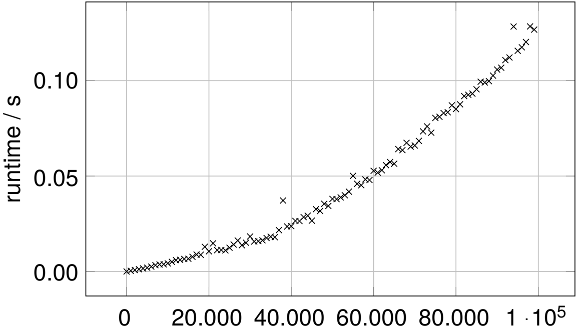


Figure: Summing elements in random order

Cache Efficiency

Algorithm Comparision

Conclusion:

Cache Efficiency
Algorithm Comparision

Conclusion:

- The number of operations are identical for both algorithms

Cache Efficiency

Algorithm Comparision

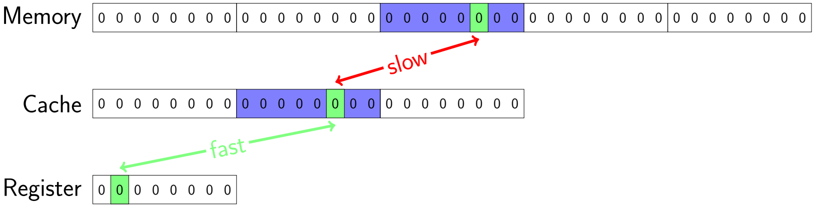
Conclusion:

- 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

Structure

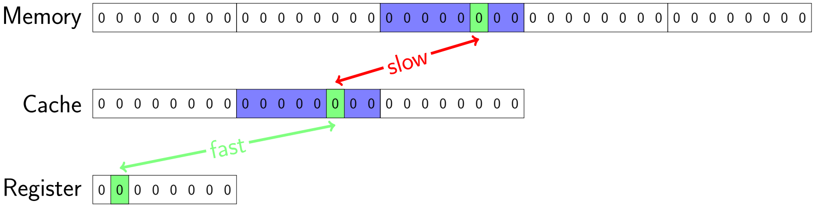
Cache Efficiency

CPU Cache



Cache Efficiency

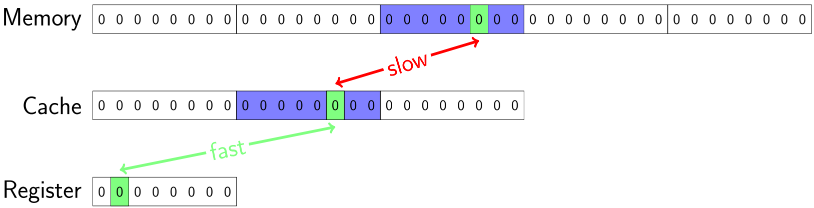
CPU Cache



Principle / organization:

Cache Efficiency

CPU Cache

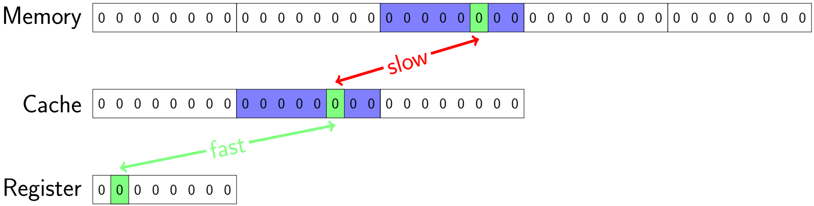


Principle / organization:

- Accessing one byte of the main memory takes $\approx 100\,\text{ns}$

Cache Efficiency

CPU Cache

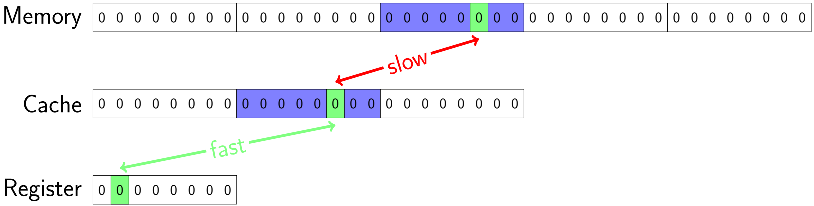


Principle / organization:

- Accessing one byte of the main memory takes ≈ 100 ns
- Accessing one byte of (L1-)cache takes ≈ 1 ns

Cache Efficiency

CPU Cache

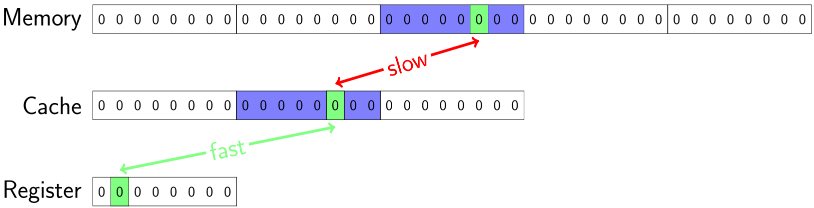


Principle / organization:

- Accessing one byte of the main memory takes $\approx 100\,\text{ns}$
- Accessing one byte of (L1-)cache takes $\approx 1\,\text{ns}$
- Accessing one or more byte/s of main memory loads a whole block $\approx 100\,\text{B}$ into the cache

Cache Efficiency

CPU Cache

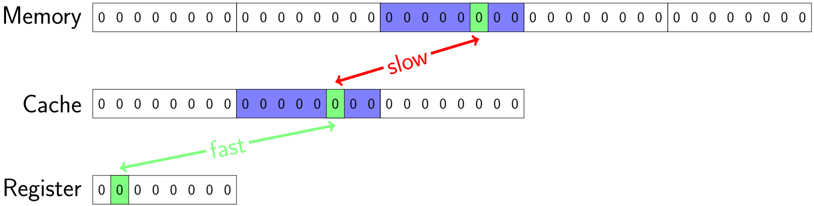


Principle / organization:

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

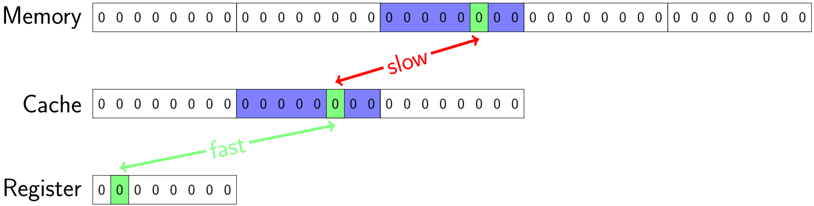
Cache Efficiency

CPU Cache



Cache Efficiency

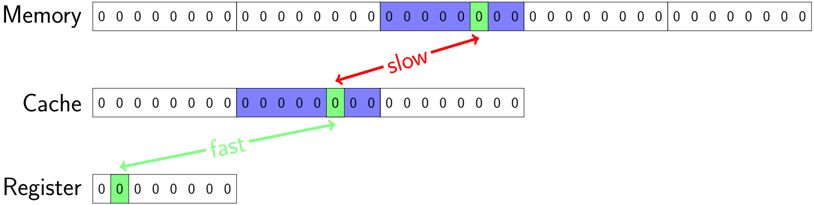
CPU Cache



Cache organization:

Cache Efficiency

CPU Cache

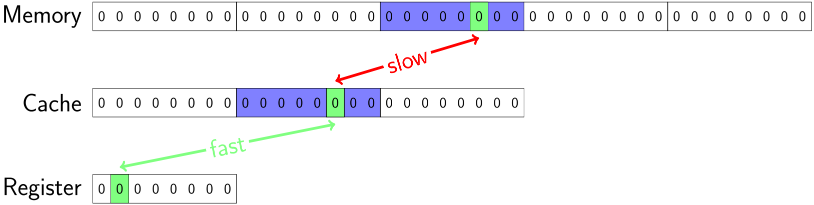


Cache organization:

- The (L1-)cache can hold multiple memory blocks

Cache Efficiency

CPU Cache

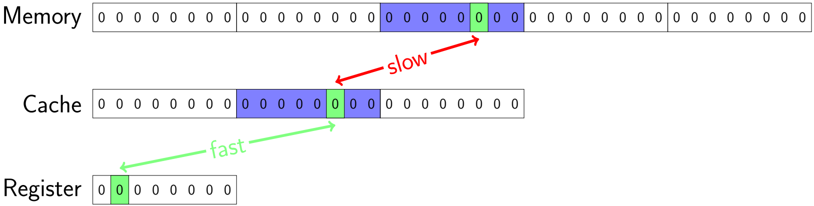


Cache organization:

- The (L1-)cache can hold multiple memory blocks
  - Cache lines ≈ 100 kB

Cache Efficiency

CPU Cache

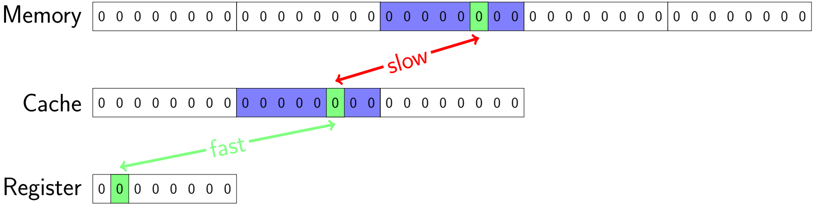


Cache organization:

- The (L1-)cache can hold multiple memory blocks
  - Cache lines $\approx 100\,\text{kB}$
- If the capacity is reached unused blocks are discarded

Cache Efficiency

CPU Cache

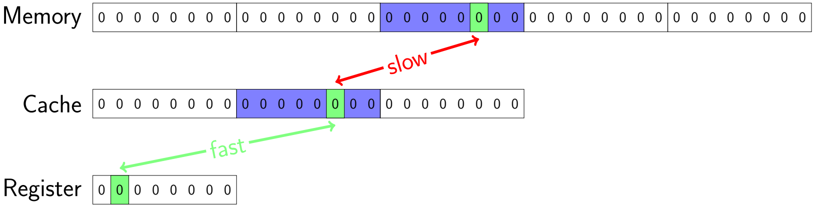


Cache organization:

- The (L1-)cache can hold multiple memory blocks
  - Cache lines ≈ 100 kB
- If the capacity is reached unused blocks are discarded
  - Least recently used (LRU)

Cache Efficiency

CPU Cache

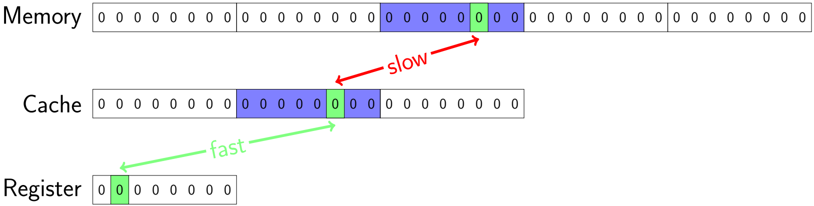


Cache organization:

- The (L1-)cache can hold multiple memory blocks
  - Cache lines ≈ 100 kB
- If the capacity is reached unused blocks are discarded
  - Least recently used (LRU)
  - Least frequently used (LFU)

Cache Efficiency

CPU Cache

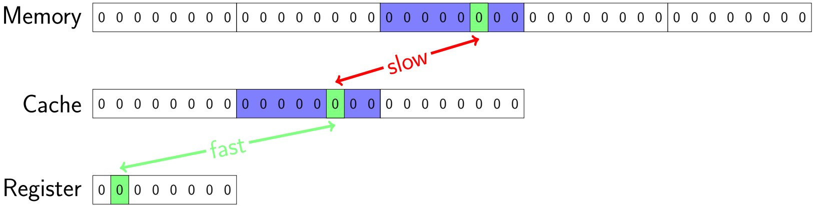


Cache organization:

- The (L1-)cache can hold multiple memory blocks
  - Cache lines $\approx 100\,\mathrm{kB}$
- If the capacity is reached unused blocks are discarded
  - Least recently used (LRU)
  - Least frequently used (LFU)
  - First in first out (FIFO)

Cache Efficiency

CPU Cache

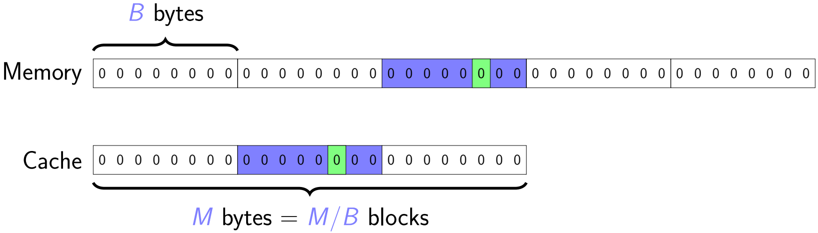


Cache organization:

- The (L1-)cache can hold multiple memory blocks
  - Cache lines $\approx 100\,\mathrm{kB}$
- 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

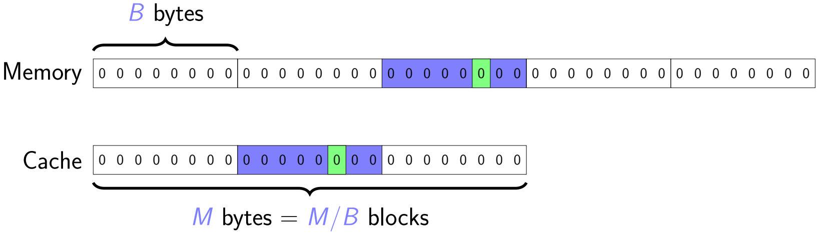
Cache Efficiency

Block Operations



Terminology:

Cache Efficiency
Block Operations

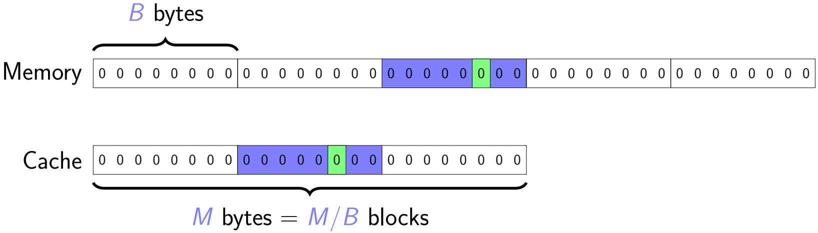


Terminology:

- The system consists of slow and fast memory

Cache Efficiency

Block Operations

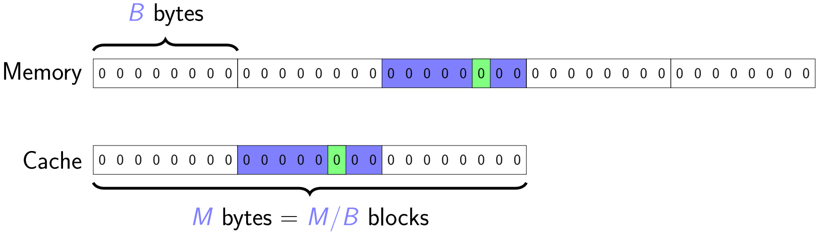


Terminology:

- The system consists of slow and fast memory
- The slow memory is divided in blocks of size B

Cache Efficiency

Block Operations

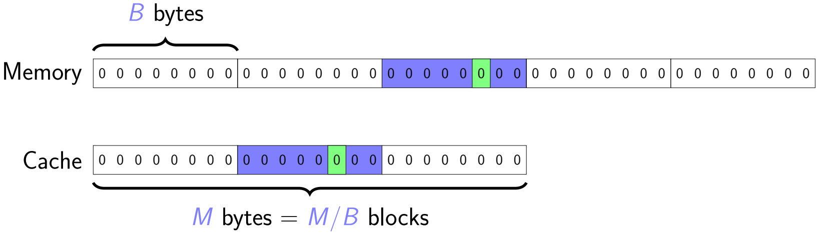


Terminology:

- The system consists of slow and fast memory
- The slow memory is divided in blocks of size B
- The fast cache has size M an can store M/B blocks

Cache Efficiency

Block Operations

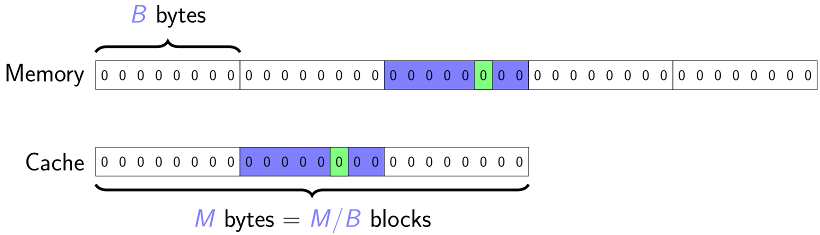


Terminology:

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

Cache Efficiency

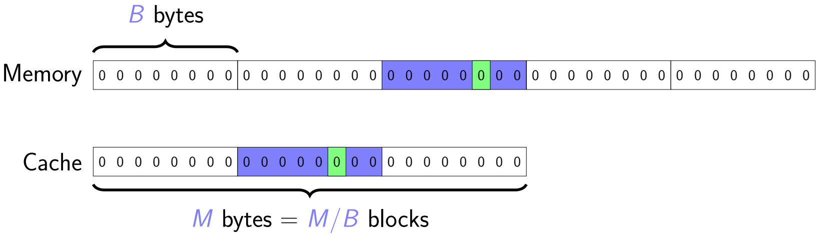
Block Operations



Terminology:

Cache Efficiency

Block Operations

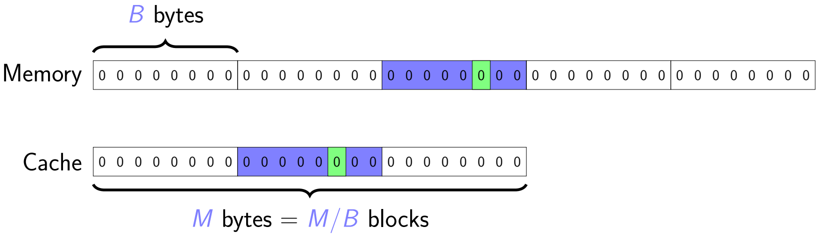


Terminology:

- The program defines which blocks are held in the cache

Cache Efficiency

Block Operations

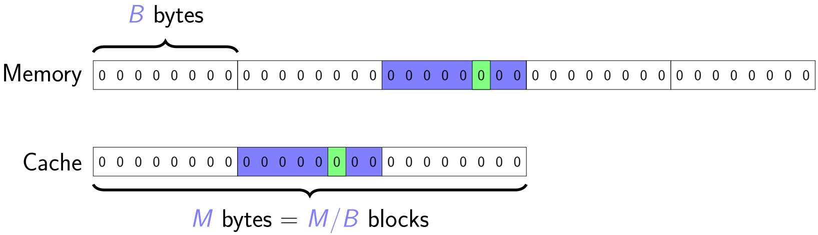


Terminology:

- The program defines which blocks are held in the cache
- We use the number of block operations as runtime estimation

Cache Efficiency

Block Operations



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

Cache Efficiency
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:

Cache Efficiency

Block Operations

Additional factors:

- The following settings change only a small constant factor in number of block operations

Cache Efficiency

Block Operations

Additional factors:

- The following settings change only a small constant factor in number of block operations
  - Partionining of the slow memory into blocks

Cache Efficiency
Block Operations

Additional factors:

- The following settings change only a small constant factor in number of block operations
  - Partionining of the slow memory into blocks
  - Regardless of the block size: 1 Bytes or 4 Bytes or 8 Bytes

Cache Efficiency

Block Operations

Additional factors:

- The following settings change only a small constant factor in number of block operations
  - Partionining of the slow memory into blocks
  - Regardless of the block size: 1 Bytes or 4 Bytes or 8 Bytes

Note:

Cache Efficiency

Block Operations

Additional factors:

- The following settings change only a small constant factor in number of block operations
  - Partionining of the slow memory into blocks
  - Regardless of the block size: 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 Efficiency

Block Operations

Additional factors:

- The following settings change only a small constant factor in number of block operations
  - Partionining of the slow memory into blocks
  - Regardless of the block size: 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

Cache Efficiency

Block Operations

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

Cache Efficiency

Block Operations

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

- CPU L1 Cache: $B = 64\,\mathrm{B}$, $M = 4 \times (32\,\mathrm{kB} + 32\,\mathrm{kB})$

Cache Efficiency

Block Operations

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

- CPU L1 Cache: $B = 64\,\mathrm{B}$, $M = 4 \times (32\,\mathrm{kB} + 32\,\mathrm{kB})$
- CPU L2 Cache: $B = 64\,\mathrm{B}$, $M = 4 \times 256\,\mathrm{kB}$

Cache Efficiency

Block Operations

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

- CPU L1 Cache: $B = 64\,\mathrm{B}$, $M = 4 \times (32\,\mathrm{kB} + 32\,\mathrm{kB})$
- CPU L2 Cache: $B = 64\,\mathrm{B}$, $M = 4 \times 256\,\mathrm{kB}$
- CPU L3 Cache: $B = 64\,\mathrm{B}$, $M = 8\,\mathrm{MB}$
- Disk Cache: $B = 64\,\mathrm{kB}$, $M = 64\,\mathrm{MB}$

Cache Efficiency

Block Operations

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

- 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

Cache Efficiency
Block Operations

Terminology:

Cache Efficiency
Block Operations

Terminology:

- Block loads on CPU-cache are called cache misses

Cache Efficiency
Block Operations

Terminology:

- Block loads on CPU-cache are called cache misses
- Block operations on disk-cache are called IOs
  (input / output operations)

Cache Efficiency

Block Operations

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

Cache Efficiency

Block Operations - Linear Order

Example 1 - Linear order:

Cache Efficiency

Block Operations - Linear Order

Example 1 - Linear order:

- We sum up all elements in natural order

$$\mathrm{sum}(a) = a[1] + a[2] + \cdots + a[n]$$

Cache Efficiency

Block Operations - Linear Order

Example 1 - Linear order:

- We sum up all elements in natural order

$$\mathrm{sum}(a) = a[1] + a[2] + \cdots + a[n]$$

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

Cache Efficiency

Block Operations - Linear Order

Example 1 - Linear order:

- We sum up all elements in natural order

$$\mathrm{sum}(a) = a[1] + a[2] + \cdots + a[n]$$

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

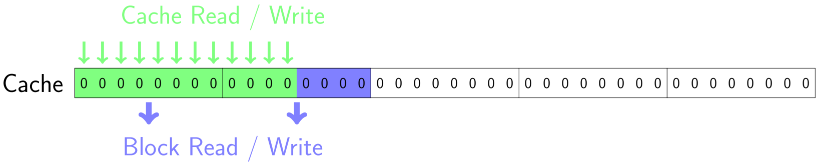


Figure: Good locality of sum operation

Example 2 - Random order:

Cache Efficiency

Block Operations - Random Order

Example 2 - Random order:

- We sum up all elements in random order

$$\mathrm{sum}(a) = a[21] + a[5] + \cdots + a[8]$$

Example 2 - Random order:

- We sum up all elements in random order

$$\mathrm{sum}(a) = a[21] + a[5] + \cdots + a[8]$$

- The number of block operations is n in the worst case

Example 2 - Random order:

- We sum up all elements in random order

$$\mathrm{sum}(a) = a[21] + a[5] + \cdots + a[8]$$

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

Cache Efficiency

Block Operations - Random Order

Example 2 - Random order:

- We sum up all elements in random order

$$\mathrm{sum}(a) = a[21] + a[5] + \cdots + a[8]$$

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

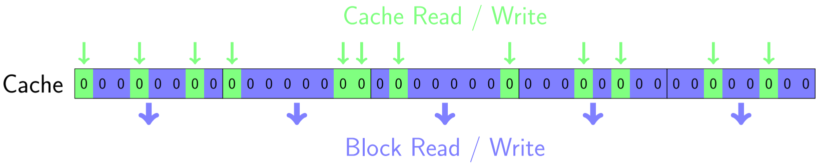


Figure: Bad locality of sum operation

Cache Efficiency

Block Operations

Generally the factor is substantially $< B$

Cache Efficiency

Block Operations

Generally the factor is substantially $< B$

- Even with a random order we access 4 neighboring bytes at once per `int` (int32_t)

Cache Efficiency

Block Operations

Generally the factor is substantially $< B$

- Even with a random order we access 4 neighboring bytes at once per `int` (int32_t)
- The next element might already be loaded in the cache

Cache Efficiency
Block Operations

Generally the factor is substantially $< B$

- Even with a random order we access 4 neighboring bytes at once per `int` (int32_t)
- The next element might already be loaded in the cache
- If not $n \gg M$ this might occur with a high probability

Cache Efficiency

Block Operations - QuickSort

QuickSort:

Cache Efficiency

Block Operations - QuickSort

QuickSort:

- **Strategy:** Divide and conquer

Cache Efficiency

Block Operations - QuickSort

QuickSort:

- **Strategy:** Divide and conquer
- Divide the data into two parts where the "left" part contains all values $\leq$ those in the right part

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

Cache Efficiency

Block Operations - QuickSort

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

Cache Efficiency

Block Operations - QuickSort

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

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

Idea of Quicksort

- **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)

Idea of Quicksort

- **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)

Idea of Quicksort

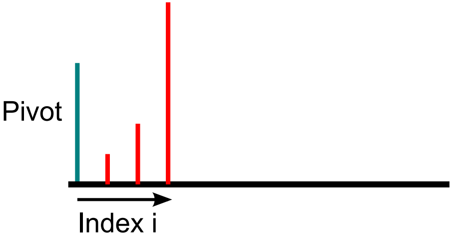
- **At start:** Pivot in first position, first re-arrange list such that left part contains small, right part larger elements
- Do required changes *in place*

- **End point:** k is left to left-most element greater than pivot

  swap position 0 (pivot) with k (smaller than pivot)

Idea of Quicksort

- **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*

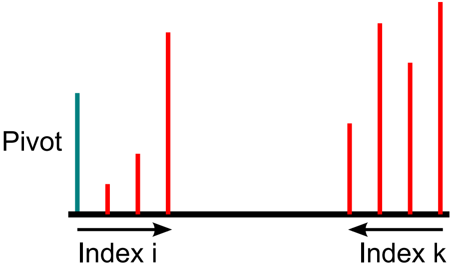


- **End point:** k is left to left-most element greater than pivot

  swap position 0 (pivot) with k (smaller than pivot)

Idea of Quicksort

- **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*

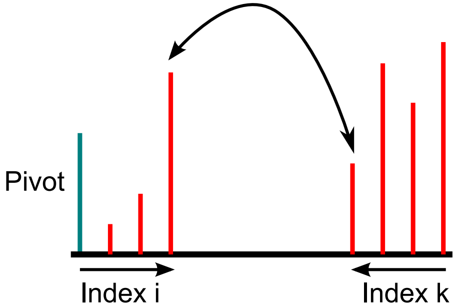


- **End point:** k is left to left-most element greater than pivot

  swap position 0 (pivot) with k (smaller than pivot)

Idea of Quicksort

- **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*

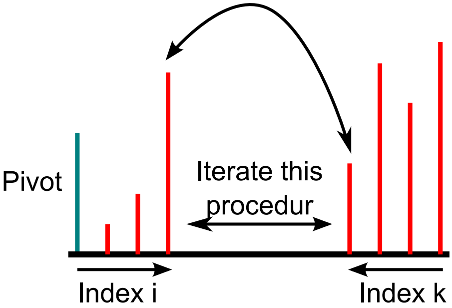


- **End point:** k is left to left-most element greater than pivot

  swap position 0 (pivot) with k (smaller than pivot)

Idea of Quicksort

- **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)

Cache Efficiency

Block Operations - QuickSort - Python

Python:

```
def quicksort(l, start, end):
  if (end - start) < 1:
    return

  i = start
  k = end
  piv = l[0]

  ...
```

Cache Efficiency

Block Operations - QuickSort - Python

```python
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:

Number of operations for Quicksort:

- Let $T(n)$ be the runtime for the input size n

Number of operations for Quicksort:

- Let $T(n)$ be the runtime for the input size n

Assumptions:

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

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 $k = \log_2 n$

Cache Efficiency

Block Operations - QuickSort

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

Cache Efficiency

Block Operations - QuickSort

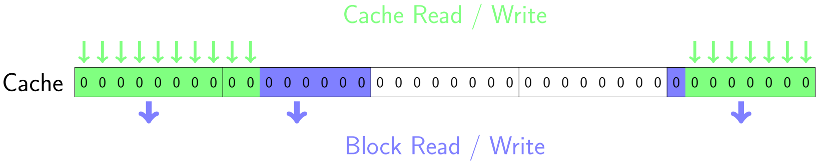


Figure: Locality of quicksort

Cache Efficiency

Block Operations - QuickSort

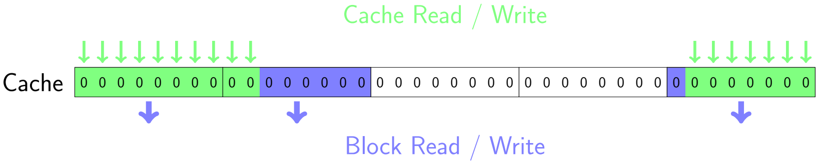


Figure: Locality of quicksort

- Let $IO(n)$ be the number of block operations for input size n

Cache Efficiency

Block Operations - QuickSort

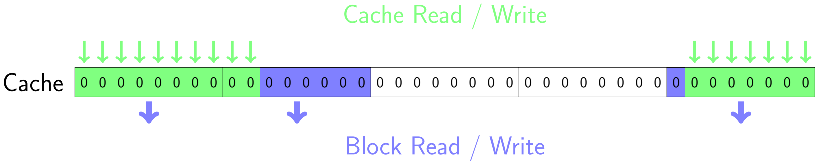


Figure: Locality of quicksort

- 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?

Cache Efficiency

Block Operations - QuickSort

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

Structure

Concept:

Divide and Conquer

Introduction

Concept:

- Divide the problem into smaller subproblems

Divide and Conquer

Introduction

Concept:

- Divide the problem into smaller subproblems
- Conquer the subproblems through recursive solving.
  If subproblems are small enough solve them directly

Divide and Conquer

Introduction

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

Divide and Conquer

Introduction

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

Divide and Conquer

Introduction

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 sufficently small subproblems

Divide and Conquer

Introduction - Python

Divide and Conquer

Introduction - Python

- Function solve for solving a problem of size n

Divide and Conquer

Introduction - Python

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

```
def solve(problem):
    if n < threshold:
        return solution # solve directly
    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
```

Divide and Conquer

Features

Divide and Conquer:

Divide and Conquer

Features

Divide and Conquer:

- Can help with conceptual hard problems

Divide and Conquer:

- Can help with conceptual hard problems
- Solution of the trivial problems has to be known

Divide and Conquer

Features

Divide and Conquer:

- Can help with conceptual hard problems
- Solution of the trivial problems has to be known
- Dividing in subproblems has to be possible

Divide and Conquer

Features

Divide and Conquer:

- 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

Divide and Conquer

Features

Features:

- Realization of efficient solutions

Features:

- Realization of efficient solutions
  - If trivial solution is $\in O(1)$

Divide and Conquer

Features

Features:

- Realization of efficient solutions
  - If trivial solution is $\in O(1)$
  - And separation / combination of subproblems is $\in O(n)$

Divide and Conquer

Features

Features:

- 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

Divide and Conquer

Features

Features:

- 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 $\in O(n \cdot \log n)$

Divide and Conquer
Features

Features:

- 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

Divide and Conquer

Features

Features:

- 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

Features:

- 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:

Definition of the trivial case:

- Smaller subproblems are elegant and simple

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

Divide and Conquer

Implementation

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:

Divide and Conquer

Implementation

Division in subproblems:

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

Divide and Conquer

Implementation

Division in subproblems:

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

Combination of solutions:

Division in subproblems:

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

Combination of solutions:

- Typically conceptional demanding

Example - Maximum Subtotal

Example - Maximum Subtotal Input:

Example - Maximum Subtotal Input:

Example - Maximum Subtotal Input:

- Sequence X of n integers

Example - Maximum Subtotal Input:

- Sequence X of n integers

Output:

Example - Maximum Subtotal Input:

- Sequence X of n integers

Output:

- Maximum sum of related subsequence and its index boundary

Example - Maximum Subtotal Input:

- Sequence 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

Example - Maximum Subtotal Input:

- Sequence 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

Divide and Conquer

Example - Maximum Subtotal

Application:

- Maximum profit of buying and selling shares

Figure: Stock value over time

Naive solution (brute force)

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

Runtime - Upper bound

```python
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:

Upper bound:

- Three interleaved loops

Divide and Conquer

Example - Maximum Subtotal

Upper bound:

- Three interleaved loops
- Each loop with runtime $O(n)$

Upper bound:

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

Lower bound:

Table: Operations

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

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

Divide and Conquer

Example - Maximum Subtotal - Runtime

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

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

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:

Runtime:

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

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

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:

Current approach:

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

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

Divide and Conquer

Example - Maximum Subtotal - Runtime

Current approach:

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

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

Better approach:

Divide and Conquer

Example - Maximum Subtotal - Runtime

Current approach:

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

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

Better approach:

- Incremental sum instead of loop

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

Divide and Conquer

Example - Maximum Subtotal - Python

Better solution:

Divide and Conquer

Example - Maximum Subtotal - Python

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
```

Better solution:

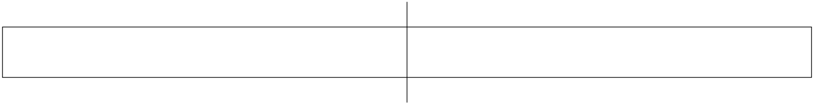
```
def maxSubArray(X):
    result = (X[0], 0, 0)
    # n loops -> O(n)
    for i in range(0, len(X)):
        subSum = 0
        # max n loops -> O(n)
        for j in range(i, len(X)):
            subSum += X[j] # O(1)
            if result[0] < subSum: # O(1)
                result = (subSum, i, j)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

- Runtime $\in O(n^2)$

Divide and Conquer

Example - Maximum Subtotal

Divide and Conquer:



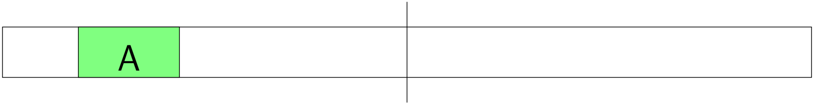
Divide and Conquer Idea to solve:

- Split the sequence in the middle

Divide and Conquer

Example - Maximum Subtotal

Divide and Conquer:



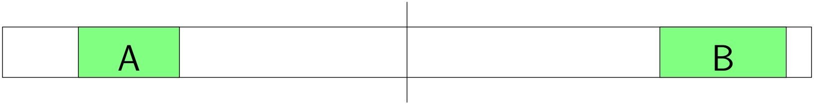
Divide and Conquer Idea to solve:

- Split the sequence in the middle
- Solve left half of the problem

Divide and Conquer

Example - Maximum Subtotal

Divide and Conquer:



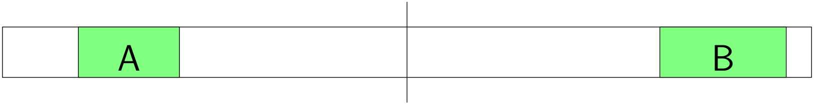
Divide and Conquer Idea to solve:

- Split the sequence in the middle
- Solve left half of the problem
- Solve right half and combine both solutions into one

Divide and Conquer

Example - Maximum Subtotal

Divide and Conquer:



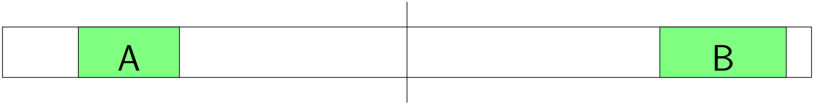
Divide and Conquer Idea to solve:

- Split the sequence in the middle
- Solve left half of the problem
- Solve right half and combine both solutions into one
- Maximum might be located in left half (A) or right half (B)

Divide and Conquer

Example - Maximum Subtotal

Divide and Conquer:



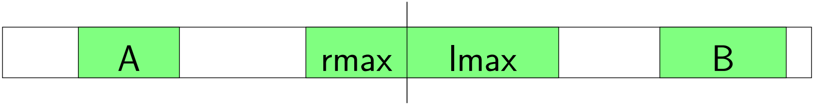
Divide and Conquer Idea to solve:

- Split the sequence in the middle
- Solve left half of the problem
- Solve right half and combine both solutions into one
- Maximum might be located in left half (A) or right half (B)
- Problem: Maximum can overlap the split

Divide and Conquer

Example - Maximum Subtotal

Divide and Conquer:



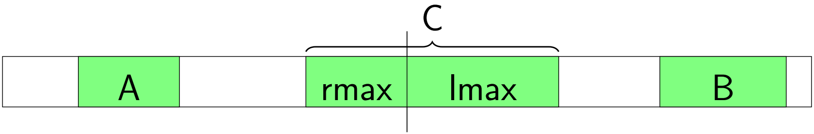
Divide and Conquer Idea to solve:

- Split the sequence in the middle
- Solve left half of the problem
- Solve right half and combine both solutions into one
- Maximum might be located in left half (A) or right half (B)
- Problem: Maximum can overlap the split
- To solve this case we have to calculate rmax and lmax

Divide and Conquer

Example - Maximum Subtotal

Divide and Conquer:



Divide and Conquer Idea to solve:

- Split the sequence in the middle
- Solve left half of the problem
- Solve right half and combine both solutions into one
- Maximum might be located in left half (A) or right half (B)
- Problem: Maximum can overlap the 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:

Principle - Divide and Conquer:

- Small problems are solved directly: $n = 1 \Rightarrow \text{max} = X[0]$

Principle - Divide and Conquer:

- Small problems are solved directly: $n = 1 \Rightarrow \max = X[0]$
- Bigger problems are partitioned into two subproblems and recursively solved. Subsolutions A and B are returned

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

Principle - Divide and Conquer:

- Small problems are solved directly: $n = 1 \Rightarrow \max = X[0]$
- 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

Example - Maximum Subtotal - Python

```
def maxSubArray(X, i, j):
    if i == j: # trivial case
        return (X[i], i, i)

    # recursive subsolutions for A, B
    m = (i + j) / 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    # rmax and lmax for cornercase C
    C1, C2 = rmax(X, i, m), lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    # compute solution from results A, B, C
    return max([A, B, C], key=lambda i: i[0])
```

Further Literature

- **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.

Further Literature

- **Caching**

[Wik] Cache
https://en.wikipedia.org/wiki/Cache