32c ⟨* 30a⟩+≡ ◁32b

①

```
void main()
{
  class ext_stack<float> st(Ändreas");

  for(int i=0;i<1290000;i++) st.push(i);
  int help = st.size();
  for(int j=0;j<help;j++)
    {
      if(st.pop() != help-j-1)
        {
          cout << ËRROR pop\n";
          exit(1);
        }
    }
  cout << SSize of Stack : " << st.size();
}
```

Uses **ext_stack** 31, **pop** 32, and **push** 32.

3 The paging problem (by Dr. Susanne Albers)

In this section we study an important problem operating systems are faced with when managing main memory: At any time, an operating system has to decide which data to store in main memory. More precisely, whenever a program has to access a data block that is not in main memory, the operating system must remove a data block (from main memory) in order to make room for the requested block. The problem which block to evict is known as the *paging problem*.

In the standard paging terminology, a data block is called a *page*. We begin with a formal definition of the paging problem.

The paging problem: Consider a two-level memory system that consists of a small main memory and a large secondary memory. A sequence of *requests* to pages in the memory system must be served by a paging algorithm. A request is *served* if the corresponding page is in main memory. If a requested page is not in main memory, a *page fault* occurs. Then a page must be moved from main memory to secondary memory so that the requested page can be loaded into the vacated location. A paging algorithm specifies which page to evict on a fault. The goal is to minimize the total number of page faults incurred on the entire request sequence.

Paging is an *online problem*, i.e., the decision which page to evict on a fault must be made without knowledge of any future requests.

We list a number of well-known paging algorithms.

- **LRU** (Least Recently Used): On a fault, evict the page in main memory that was requested least recently.
- **FIFO** (First-In First-Out): Evict the page that has been in main memory longest.
- **RANDOM**: Evict a page that is chosen uniformly at random from among the pages in main memory
- **LFU** (Least Frequently Used): Evict the page that has been requested least frequently.

All of these algorithms are *demand paging algorithms*, i.e., they only evict a page if there is a page fault.

There are two important methods for analyzing the performance of a paging algorithm.

> *Worst Case Analysis:* A paging algorithm is analyzed on an arbitrary request sequence that may be generated by an adversary. This type of analysis is also known as *competitive analysis.*
>
> *Average Case Analysis*: A paging algorithm is analyzed on request sequences that are generated by probability distributions.

In the following we first concentrate on competitive analysis.

3.1 Competitive analysis

3.1.1 Deterministic paging

Let $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ be a request sequence. Each request $\sigma(t)$, $1 \leq t \leq m$, specifies one page in the memory system. Given a request sequence σ and an online paging algorithm A, let

$$F_A(\sigma)$$

be the number of page fault incurred by A on σ. Furthermore, let

$$F_{OPT}(\sigma)$$

be the number of page faults incurred by an *optimal offline* algorithm. An optimal offline algorithm knows the entire request sequence σ in advance and can serve it with the minimum number of page faults.

1. Definition
An online paging algorithm A is called *c-competitive* if there exists a constant a such that

$$F_A(\sigma) \leq c \cdot F_{OPT}(\sigma) + a$$

for all request sequences σ. Note that competitive analysis is a very strong performance measure. An online algorithm is compared to an "all-knowing" offline algorithm. A competitive algorithm has to perform well on all possible request sequences.

In the case of the paging problem, an optimal offline algorithms can be described explicitly. The algorithm is called the MIN algorithm and was discovered by Belady [12].

> **MIN**: On a fault, evict the page whose next request occurs farthest in the future.

We will prove later that on any sequence of requests, MIN achieves the minimum number o page faults.

Throughout this section, let k denote the number of pages that can simultaneously reside ir main memory. The following two theorems are due to Sleator and Tarjan [51].

Theorem 1 *The algorithm LRU is k-competitive.*

Proof: Consider an arbitrary request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$. We will prove that $F_{LRU}(\sigma) \leq k \cdot F_{OPT}(\sigma)$. Without loss of generality we assume that LRU and OPT initially start with the same main memory.

We partition σ into phases $P(0), P(1), P(2), \ldots$ such that LRU has at most k fault on $P(0)$ and exactly k faults on $P(i)$, for every $i \geq 1$. Such a partitioning can be obtained easily. We start at the end of σ and scan the request sequence. Whenever we have seen k faults made by LRU, we cut off a new phase. In the remainder of this proof we will show that OPT has at least one page fault during each phase. This establishes the desired bound.

For phase $P(0)$ there is nothing to show. Since LRU and OPT start with the same main memory, OPT has a page fault on the first request on which LRU has a fault.

Consider an arbitrary phase $P(i)$, $i \geq 1$. Let $\sigma(t_i)$ be the first request in $P(i)$ and let $\sigma(t_{i+1} - 1)$ be the last request in $P(i)$. Furthermore, let x be the page that is requested last in $P(i-1)$.

Lemma 1 *$P(i)$ contains requests to k distinct pages that are different from x.*

If the lemma holds, then OPT must have a page fault in $P(i)$. OPT has page x in its main memory at the end of $P(i-1)$ and thus cannot have all the other k pages request in $P(i)$ in its main memory.

It remains to prove the lemma. The lemma clearly holds if the k requests on which LRU has a fault are to k distinct pages and if these pages are also different from x. So suppose that LRU faults twice on a page y in $P(i)$. Assume that LRU has a fault on $\sigma(s_1) = y$ and $\sigma(s_2) = y$, with $t_i \leq s_1 < s_2 \leq t_{i+1} - 1$. Page y is in LRU's main memory immediately after $\sigma(s_1)$ is served and is evicted at some time t with $s_1 < t < s_2$. When y is evicted, it is the least recently requested page in main memory. Thus the subsequence $\sigma(s_1), \ldots, \sigma(t)$ contains requests to $k+1$ distinct pages, at least k of which must be different from x.

Finally suppose that within $P(i)$, LRU does not fault twice on page but on one of the faults, page x is request. Let $t \geq t_i$ be the first time when x is evicted. Using the same arguments as above, we obtain that the subsequence $\sigma(t_i - 1), \sigma(t_i), \ldots, \sigma(t)$ must contain $k+1$ distinct ~~pages~~ [illegible]

The next theorem implies that LRU achieves the best possible competitive ratio.

Theorem 2 *Let A be a deterministic online paging algorithm. If A is c-competitive, then $c \geq k$.*

Proof: Let $S = \{x_1, x_2, \ldots, x_{k+1}\}$ be a set of $k+1$ arbitrary pages. We assume without loss of generality that A and OPT initially have $x_1, \ldots, x_k$ in their main memories.

Consider the following request sequence. Each request is made to the page that is not in A's fast memory.

Online algorithm A has a page fault on every request. Suppose that OPT has a fault on some request $\sigma(t)$. When serving $\sigma(t)$, OPT can evict a page is not requested during the next $k-1$ requests $\sigma(t+1), \ldots, \sigma(t+k-1)$. Thus, on any k consecutive requests, OPT has at most one fault.

~~**Theorem 3** *On every request sequence σ, MIN incurs the minimum number of page faults.*~~

~~**Proof:** Consider an arbitrary request sequence $\sigma = \sigma(1), \sigma(2), \ldots, \sigma(m)$ and let A be an other algorithm for serving σ. Suppose that A and MIN start in the same initial state and serve the first t requests $\sigma(1), \ldots, \sigma(t)$ identically. We show that A can be modified so that A and MIN serve the first $t+1$ requests identically and the number of page faults made by A does not increase. Repeated execution of this step shows that A can be transformed so that it behaves in exactly the same way as MIN, and the number of page faults does not increase during the transformation.~~

~~Let x be the page requested by $\sigma(t+1)$. Suppose that MIN serves $\sigma(t+1)$ by evicting u from its main memory and that A serves $\sigma(t+1)$ by evicting v, $v \neq u$.~~

~~Define algorithm A' as follows. A' serves $\sigma(t+1)$ by evicting u and then works in the same [illegible] until one of the following events occurs.~~

Case 1: There is a page fault at a request to page y, $y \neq v$, and A evicts page u. In this case, A' evicts page v. At this point, A and A' are again in the same state and incurred the same number of page faults.

Case 2: There is a page fault at a request to page v, and A evicts a page z. In this case, A' evicts page z and loads u. Again, after this operation, A and A' are in the same state and had the same number of page faults.

Note that, by the definition of MIN, a request to u cannot occur earlier than a request to v. ∎

3.1.2 Randomized paging

A natural question is: Can we improve the competitive ratio of k using randomization? We first have to define the competitive ratio of a randomized paging algorithm.

1. Definition

A randomized online paging algorithm is called *c-competitive* if there is a constant a such that

$$E[F_A(\sigma)] \leq c \cdot F_{OPT}(\sigma) + a$$

for all request sequences σ. Here, $E[F_A(\sigma)]$ is the expected number of pages faults incurred by A, where the expectation is taken of the random choices made by A.

Let us consider the RANDOM algorithms introduced in the beginning of this section. Raghavan and Snir [48] proved the following theorem.

Theorem 4 *The RANDOM algorithm is not better than k-competitive.*

Proof: We consider the request sequence

$$\sigma = x_1 x_2 x_3 \ldots x_k (y_1 x_2 x_3 \ldots, x_k)^l (y_2 x_2 x_3 \ldots, x_k)^l (y_3 x_2 x_3 \ldots, x_k)^l \ldots$$

The pages $x_1, \ldots, x_k, y_1, y_2, \ldots$ are pairwise distinct. Here $(\rho)^l$ denotes the l-fold repetition of the subsequence ρ, where l is a positive integer to be specified later.

Clearly, OPT has exactly one page fault in each subsequence $(y_i x_2 x_3 \ldots, x_k)^l$.

At the beginning of each subsequence $\rho_i = (y_i x_2 x_3 \ldots, x_k)^l$, RANDOM has at most $k - 1$ of the pages requested in ρ_i in its main memory. We say that RANDOM has a *near page fault* on ρ_i if it has exactly $k - 1$ of the k pages requested in ρ_i in its main memory and a page fault occurs. We say that RANDOM has *success on an near fault* if it evicts the page not requested in ρ_i.

On each repetition of the sequence $y_i x_2 x_3 \ldots, x_k$, RANDOM has at least one page fault until it has success on a near fault. The probability that RANDOM has success on a near fault is $\frac{1}{k}$, i.e, the expected number of near faults until there is success on a near fault [illegible] κ. is Choosing l large enough we can make the ratio $E(F_{RANDOM}(\sigma)/F_{OPT}(\sigma))$ arbitrarily close to k.
∎

The above theorem implies that in order to beat the competitive ratio of k, randomization has to be used in a more sophisticated way. The following algorithms was analyzed by Fiat *et al.* [28].

Algorithm MARKING: The algorithm processes a request sequence in phases. At the beginning of each phase, all pages in the memory system are unmarked. Whenever a page is requested, it is *marked*. On a fault, a page is chosen uniformly at random from among the unmarked pages in main memory, and this pages is evicted. A phase ends when all pages in fast memory are marked and a page fault occurs. Then, all marks are erased and a new phase is started.

[illegible]

Theorem 5 *The MARKING algorithm is $2H_k$-competitive, where $H_k = \sum_{i=1}^{k} 1/i$ is the k-th Harmonic number.*

Note that H_k is roughly $\ln k$. Later we will see that no randomized online paging algorithm can be better than H_k-competitive. Thus the MARKING algorithm is optimal, up to a constant factor. More complicated paging algorithms achieving an optimal competitive ratio of H_k were given in [39, 1].

Proof: Given a request sequence $\sigma = \sigma(1), \ldots, \sigma(m)$, we assume without of generality that MARKING already has a fault on the first request $\sigma(1)$.

MARKING divides the request sequence into phases. A phase starting with $\sigma(i)$ ends with $\sigma(j)$, where j, $j > i$, is the smallest integer such that the set

$$\{\sigma(i), \sigma(i+1), \ldots, \sigma(j+1)\}$$

contains $k +$ ~~1~~ [illegible] distinct pages. Note that at the end of a phase all pages in main memory are marked.

Consider an arbitrary phase. Call a page *stale* if it is unmarked but was marked in the previous phase. Call a page *clean* if it is neither stale nor marked.

Let c be the number of clean pages requested in the phase. We will show that

1. the amortized number of faults made by OPT during the phase it at least $\frac{c}{2}$.
2. the expected number of faults made by MARKING is at most cH_k.

These two statements imply the theorem.

We first analyze OPT's page faults. Let S_{OPT} be the set of pages contained in OPT's main memory, and let S_M be the set of pages stored in MARKING's main memory. Furthermore, let d_I be the value of $|S_{OPT} \setminus S_M|$ at the beginning of the phase and let d_F be the value of $|S_{OPT} \setminus S_M|$ at the end of the phase. OPT has at least $c - d_I$ faults during the phase because at least $c - d_I$ of the c clean pages are not in OPT's main memory. Also, OPT has at least d_F faults during the phase because d_F pages requested during the phase are not in OPT's main memory at the end of the phase. We conclude that OPT incurs at least

$$\max\{c - d_I, d_F\} \geq \frac{1}{2}(c - d_I + d_F) = \frac{c}{2} - \frac{d_I}{2} + \frac{d_F}{2}$$

faults during the phase. Summing over all phases, the terms $\frac{d_I}{2}$ and $\frac{d_F}{2}$ telescope, except for the first and last terms. Thus the amortized number of page faults made by OPT during the phase is at least $\frac{c}{2}$.

Next we analyze MARKING's expected number of page faults. Serving c requests to clean pages cost c. There are $s = k - c \leq k - 1$ requests to stale pages. For $i = 1, \ldots, s$, we compute the expected cost of the i-th request to a stale page. Let $c(i)$ be the number of clean pages that were requested in the phase immediately before the i-th request to a stale page and let $s(i)$ denote the number of stale pages that remain before the i-th request to a stale page.

When MARKING serves the i-th request to a stale page, exactly $s(i) - c(i)$ of the $s(i)$ stale pages are in main memory, each of them with equal probability. Thus the expected cost of the request is

$$\frac{s(i) - c(i)}{s(i)} \cdot 0 + \frac{c(i)}{s(i)} \cdot 1 \leq \frac{c}{s(i)} = \frac{c}{k - i + 1}.$$

The last equation follows because $s(i) = k - (i - 1)$. The total expected cost for serving requests to stale pages is:

$$\sum_{i=1}^{s} \frac{c}{k+1-i} \leq \sum_{i=2}^{k} \frac{c}{i} = c(H_k - 1).$$

The total MARKING's expected cost in the phase is bounded by cH_k.

3.3 Paging with locality of reference

We re-consider the question why our previous analyses do not give performance bounds that are meaningful in practice.

Clearly, request sequences generated by real programs/processes are not arbitrary. They exhibit *locality of reference*: A currently requested page is likely to be referenced again in the near future. This is, for instance, due to looping, subroutines, stacks and variables used for counting. The phenomenon was already observed by Denning [24] in 1968. He also noticed that at any given time, a program always works with a relatively small set of pages that are referenced actively. This *working set* may change over time.

How can locality of reference be incorporated into our analyses? In [14, 33], the notion of *access graphs* was introduced. An access graph is a directed or undirected graph in which the nodes represent the pages in the memory system. Using access graphs we can model requested sequences generated by real programs. If a page x is requested by the current request, then the next request must be made to a page that is adjacent to x in the access graph. Note that in the analyses in Section 3.1 the underlying access graph was always the complete graph.

Given a graph G, an online paging algorithm is called c-competitive if there exists a constant a such that

$$F_A(\sigma) \le c \cdot F_{OPT}(\sigma) + a$$

for all σ that "obey" G. That is, σ must be a path in G.

We summarize the main results of the study of paging with access graphs [14, 33].

1. A detailed analysis of the best competitive ratio that can be achieved on an arbitrary but fixed graph G.
2. An analysis of the competitive ratio achieved by LRU on access graphs.
3. A new online paging algorithm called FAR. On a fault, FAR evicts the page from main memory, whose distance from the currently requested page is farthest in the access graph. It was shown that for every access graph, FAR achieves the best possible competitive ratio, up to a constant factor.

The main drawback in item 3 is that the access graph must be known by the algorithm, which is not realistic in practice.

We now turn to a recent work [29] on paging with access graphs, where the paging algorithm maintains a dynamic access graph over time. The graph is built up by the algorithm itself.

```
Algorithm DG (Dynamic Access Graph)
$V$ : Vertex set of the dynamic access graph.
$E$ : Edge set of the dynamic access graph.
$w(e)$: Weight of edge $e \in E$.
$k$: Number of pages that can reside in main memory.
$c$: Global counter.
$P$: Pointer to the current page.
$\alpha, \beta, \gamma$: Parameters (in the actual implementation $\alpha = 0.8$, $\beta = 1.5$, $\gamma = 10$).
Initialization
    Let $x$ be the pages that is requested first;
    Set $V = \{x\}$, $E = \emptyset$, $P = v$;
Main algorithm
    Let $x$ be the page pointed to by $P$;
    Let $y$, $y \neq x$, be the page requested next;
    if $y \notin V$ then add $y$ to $V$;
    Let $e = (x, y)$;
```

```
if $e \notin E$ then add $e$ to $E$; set $w(e) = 1$;
        else $w(e) = \alpha w(e)$;
if $w(e) > 1$ then $w(e) = 1$;
$c = c + 1$;
if $c = 0 (\mathrm{mod} \gamma k)$ then for all $e \in E$ set $w(e) = \beta w(e)$;
if $y$ is not in main memory then
    if main memory has no free position then
        Evict page in main memory that has the largest distance from $y$ in $(V, E, w)$;
    Load $y$;
Set $P = y$;
```

Observations:

- If a request to page x is immediately followed by a request to y, then the edge (x, y) has weight at most 1. The weight is smaller if successive requests to x and y already occurred very often.
- If pages x and y are not requested successively for a very long time, then the weight of (x, y) increases.
- The edge *weights* are crucial for deciding, which page to evict on a page fault.

Experimental tests: Fiat and Rosen tested the new algorithm DG on long request sequences consisting of 200000 to 3000000 individual requests. In particular, they compared the performace of DG to that of LRU. On average, the number of pages faults made by DG was 7 to 9 percent less than that made by LRU. In some cases, improvements of even 10 to 20 percent were observed.

We analyze a variant of the algorithm where the weight of an edge is never decreased below $\frac{1}{k}$. That is, if an edge has weight $\frac{1}{k}$, it is not decreased any further. In experimental tests, this algorithm has the same behavior as the original algorithm.

Theorem 8 *The modified algorithm DG is $O(k \log k)$-competitive.*

Proof: Given a request sequence σ, we partition σ into phases such that each phase (except for possibly the last phase) contains exactly

$$2k\gamma(\log_\beta k + 1)$$

page faults made by DG.

Consider an arbitrary phase. If this phase contains requests to at least $k + 1$ distinct pages, then OPT must have at least one page fault in the phase, and we are done. In the following we assume that the phase contains only i, $i \leq k$, page faults.

First, we concentrate on the last $2k\gamma$ page faults made by DG in the phase. There must be one page x on which DG faults twice because only $i \leq k$ pages are requested in the phase. Let t_1 and t_2, $t_1 < t_2$, be the last two times in the phase when DG has a fault on x. Let s, $t_1 < s < t_2$, be the times when x is evicted and suppose that the time interval $[t_1, s]$ contains $lk\gamma$ requests, for some integer l. At time s. the distance between x at the current pointer is at most

$$k\beta^l.$$

This is because there are at most k edges between x and the current pointer and the edge weights were set to 1 at the last traversal.

Let y be a page that is not requested in the phase. At time s, let P be the shortest path between y and the current pointer and let e be the edge on P leading into y. Let w be the weight of e at the beginning of the phase. We know $w \geq \frac{1}{k}$. At time s, the weight of e is at least $w\beta^{2\log k + l + 1} > k\beta^l$. We have a contradiction to the fact that x is evicted at time s. ■